Design and Development of an Eclipse Schema Editor for OMS^{jp}

Diploma Thesis

Christof Schmid
<s@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Michael Grossniklaus

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

22nd July 2005

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

Nowadays, Java™ has become a widespread standard for object oriented programming and a growing number of applications is implemented in this language. For this reason, the Global Information Systems Group has chosen Java™ as the language for the API to all the systems of the OMS family. This API, named OMS^{jp}, serves as a uniform platform for OMS based application development.
Furthermore, the group has created a graphical user interface on top of OMS^{jp} for modelling and administrating OMS databases. This front-end is applicable to all OMS platforms supported by OMS^{jp}. Again, this graphical user interface has been implemented in Java™. It has been developped as an Eclipse plug-in and allows OMS databases to be managed from within an integrated development environment.
However, a schema editor for graphically editing the type and classification schema was still missing. The aim of this diploma thesis is to enrich the existing front-end with the ability to build and edit databases by means of a schema diagram, thereby minimising the user's effort to create and maintain OMS databases.

Contents

1 Introduction

The object-oriented approach to database design, and in particular the integration of behaviour into the database, has presented new challenges to researchers and developers of information systems for years. It requires new methodologies, standards and tools which integrate concepts and technologies from both the software engineering and database communities.

1.1 OMS

The Global Information Systems Group at ETH Zurich has made a great effort to develop such an object-oriented database. The product of this investment is OMS, a suite of tools and technologies designed to fulfil the requirements of a new generation of data management systems. It addresses applications in the commercial, engineering and scientific domains. Further information on OMS is found in [10] and [11].
Within OMS, one and the same information model supports all the development stages from conceptual modelling to implementation. A central aspect of OMS is that its database schema plays a dual role in the development process. The schema not only describes the application domain in terms of a conceptual model, but also the representation and behaviour of the entities stored within the database. The OMS data model implements that duality by providing two abstraction levels, one for modelling the application domain and the other for modelling the entities as they are represented in the database.
The entities are classified according to the type model and the role model. The type model represents the structure and functionality of the entities, whereas the role model semantically classifies the entities in the application domain.
The Global Information System Group has chosen the generic data model OM [13] as the underlying conceptual data model throughout the development process. The use of one single data model in the implementation phase results in a process of translation rather than mapping. On this implementation level, the group promotes application frameworks for various programming environments such as Java, Prolog and Python.
Rapid prototyping with these systems is only achieved if there is the possibility to move

freely from one system to the other. The Object Model Language (OML) is a language that standardises the way in which to interact with data. Many implementations of the OM model share this common language for data definition, manipulation and querying. Yet OML does not represent an Application Programming Interface (API).

1.2 OMS^{jp}

During the last couple of years, Java has become the standard for object-oriented programming. The most reasonable choice for the development of an API, therefore, was the use of the Java^{TM} language. OMS^{jp} is the latest member of the OMS suite. The design of OMS^{jp} is laid out to support heterogenous OMS databasees as back-ends. It has been developed in order to achieve a uniform Java interface for the work with the growing number of OMS members, such as OMS Pro or eOMS. OMS^{jp} offers services to configure the underlying platform and enables the user to query databases. Furthermore, it enables applications that have been created with one OMS platform to be ported to another platform. OMS^{jp} facilitates the prototyping and development process by allowing the user to switch freely from one application to another. At the moment however a driver is only available for the OMS Pro platform.
The performance loss resulting from transforming values from OMS to Java^{TM} is reduced by storing the transformation results in a cache. This is the main reason why OMS^{jp} performs superior compared to other existing interfaces. The API definition of OMS^{jp} is found in [12], its characterisation and specification in [7].
In order to enhance software development based on OMS^{jp}, the Global Information Systems Group has designed a front-end on top of OMS^{jp} for modelling and administrating OMS databases.

1.3 OMS^{jp} Eclipse Plugin

This front-end has been implemented as a plug-in for the Eclipse Platform. Seeing that the object model of OMS is very different from the object model used in Java, OMS^{jp} offers one single class that is used to represent all user defined objects. The front-end therefore has to map the OMS^{jp} database object into the database explorer tree structure. The result is a customised internal model based on a parent-child relationship. For a detailed description of the OMS^{jp} Eclipse Plugin, see [1] and [5].

1.4 Document Structure

We start the discussion in the second chapter by a formal description of the OMS model and analyse the strengths and weaknesses of the existing OMS Pro Editors. The third chapter shows the principles of the General Editing Framework (GEF) that we have used for the implementation of our editors. Moreover, it gives a general overview of the Schema Editors' design, whereas the fourth chapter concentrates on its concrete implementation. General information on how to integrate plug-ins into the Eclipse environment is found in [2], [3] and [5].

We conclude this thesis by illustrating the advantages of the new editors over the OMS Pro Editors and propose supplementary features for future work. Finally we give a user manual which is also available as a plug-in for the Eclipse environment.

2 Evaluation

2.1 OMS Definition

First of all, we give a definition of the OMS Type Model and the OMS Classification Model. A basic knowledge of OMS Model is indispensible in order to know how to implement a schema editor best.

2.1.1 OMS Type Model

In the OMS type model, an object type is composed of a so-called type units. A type unit consists of a name and a list of attribute triples. Each attribute triple contains an attribute name, a type and a bulk property.
The bulk property in turn specifies whether an attribute comprises one single value (uni) or multiple values (set, bag, ranking or sequence).
A type definition in the OMS Data Definitions Language looks like this:

```
type eth_person subtype of person
(  office          : string;
   activities      : set of string;
);
```

An object type can also be defined as a specialisation of one or more types. Such a sub type inherits all attributes and methods from its super type(s).

2.1.2 OMS Classification Model and Its Representation

A role classifies a group of objects. It is represented by a collection of objects and a type definition. The type definition specifies the properties that are required for an object to be part of the collection.

An element belonging to a specific collection may also have additional properties. These properties are specified by sub types of the actual collection's member type. The type of a collection is a form of constraint that ensures the classified objects to have at least the set of properties specified by that type.
Graphically, an object role is represented as a shadowed rectangle. The name of the collection is given in the unshaded part, the associated member type is specified in the shadow.

2.2 OMS Pro Type and Classification Editors

In Sect. 2.2, 2.3 and 2.4, we analyse one after another the strengths and weaknesses of both editors, the type and the classification editor. It might be confusing that occasionally one and the same issue is given in both the 'Strengths' and the 'Weaknesses' sections. This is simply because these issues have nice as well as unsatisfactory facets.

Opening the Editors There are three different ways to open an editor from within the OMS Pro main application window.

Menu Bar A right click on 'Schema' opens up a pop-up menu. The selection of the entry 'Schema Graph' opens another menu which offers the choice between collections, btypes and types.

Hot Key The Hot Keys [Ctrl + G] [Ctrl + B] and [Ctrl + T] yield the same result.

Icon An editor can as well be opened by one of the three icons in the left upper corner from the tool bar of the main application window.

2.2.1 Strengths

Grid Layout Layouting a scene with a grid means that, upon the rearrangement of the objects, an element is always moved by a fixed predefined number of pixels. Grid layouting the diagram is indispensable, otherwise it is very difficult for a user to keep the schema well-organised. The grid layout helps to put the types and collections on the same level in both x- and y-directions.

Saving and Loading a Layout In addtion, there is the functionality to load and save the layout. The layout of all the diagrams is saved in a file `layout`. It is located in the root directory of the OMS database. As we will see later in this discussion, we adopted this idea within our OMS^{jp} Schema Editors. The file `eclipseLayout.xml`, also being in the root directory, comprises the information about the location, the width and the height of each collection and type in the database.

Printing The editor holds a feature to send the contents of the view to a Postscript file or to an arbitrary printing device. The printing dialog is opened by choosing the print icon from the tool bar and selecting 'Windows Extras → Settings'.

2.2.2 Weaknesses

Swapping Within the editors, the types are visually arranged as a type graph, the collections as a classification graph. Yet swapping the editors is not solved satisfactory. Whenever a user wants to switch the editor, the former one disappears from the main application window. The same effect is seen when the user tries to edit an object. A double click on a particular element opens its corresponding OMS Pro object window, but again all other editors disappear and can only be reopened from the Windows system task bar at the bottom of the screen.

Anchors For the visualisation of sub and super object relations, the developper of an editor has to consider how and where to connect the relations to the objects. In this context, the connection point is called an *anchor*. The OMS Pro Type Editor provides only one single anchor for each type, namely the midpoint of the lower horizontal edge of the rectangle. In contrast, the OMS Pro Collection Editor provides four of them, namely each midpoint of the four edges. The anchors get updated depending on the position of two collections relative to each other. Yet even with four static anchors this behaviour pattern leads to a concentration of arrows at one single point. An improved version could calculate the anchors dynamically thereby distributing the arrows equally along the edge of an object's enclosing rectangle.

Deleting Objects When a user tries to remove a type or a collection that has sub objects, he does not get the opportunity to delete only this specific object together with its relationships. The OMS Pro Schema Editor simply removes the object and all its sub objects as a whole. This is not wanted in most cases, fortunately the user gets a warning beforehand.

Preferences Page On its preferences page, the OMS Pro GUI allows the editing of various settings for the OMS Pro Schema Editor, such as for instance the font style and its size. However, the values have to be typed in manually, without a possibility to browse for them. In addition, the editors are not updated on the new layout unless they are closed and reopened again. A helpful tool within a preference page would also be a toggle button for the visualisation of a grid and a field for varying the distance of the grid lines.

2.3 OMS Pro Object Type Editor

2.3.1 Strengths

Attributes The type attributes for all the object types are shown by the menu bar entry 'Schema → Show All Attributes'. The type attributes for a specific type are displayed by choosing 'Show Attributes' from the context menu that appears when the user clicks on a type with the right mouse button.
An attribute is displayed by its name, followed by either 'set of', 'bag of', 'ranking of', 'sequence of' or simply the empty string (single-valued attribute), and its type name. In short, the attributes' details are conveniently presented according to the declaration defined in the DDL.

Automatic Layout The pop-up menu from the 'Display' entry in the menu bar offers the possibility to toggle between a user and an automatic layout. Although the capabilities of this

automatic layout are very limited, the proposed object arrangement is often suitable as a first approach.

2.3.2 Weaknesses

Sub- and Super Types The OMS Pro Editors support establishing relations between types. A right click on a specific type and choosing 'Be Supertype of' or 'Be Subtype of' opens up another context menu from which the sub or super type can be selected. The given choices however are too restricted. The OMS Type Editor excludes all the types that are already super or sub types on an arbitrary level. Figure 2.1 depicts as an example the entries for the type 'eth_person'. As indirect super types should *not* be ignored, the type 'eth_person' must get the alternative to add 'contact' as one of its super types.
The only thing to be considered, apart from not adding any direct sub or super type twice, is the avoidance of cycles in the graph.

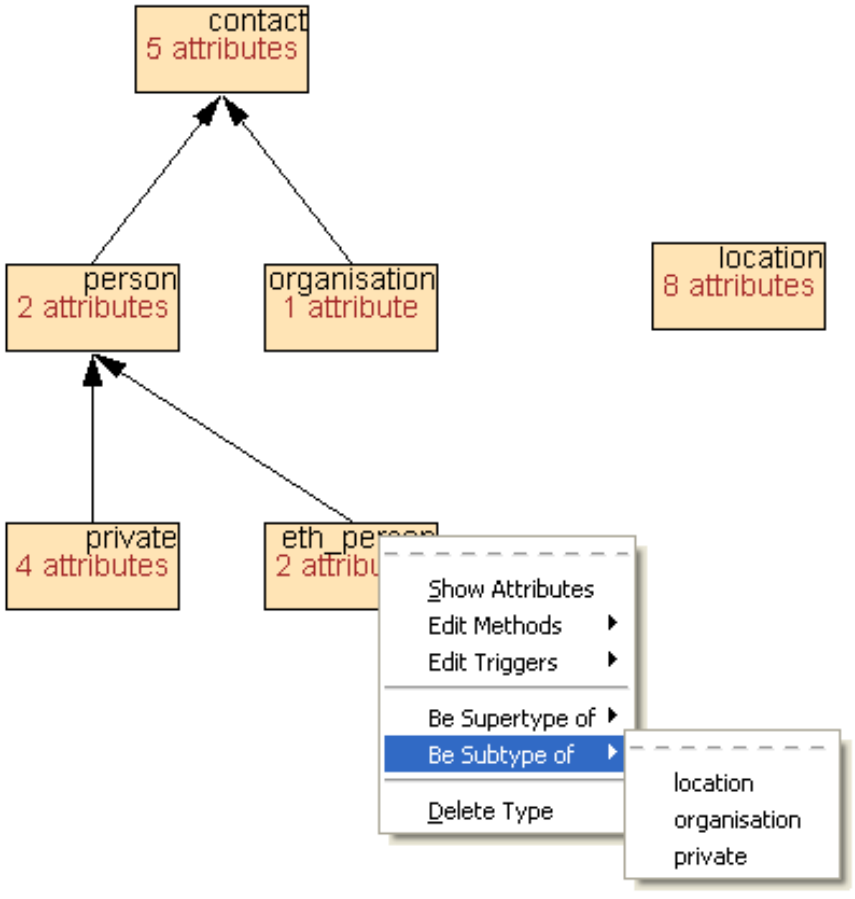

Figure 2.1: Indirect Super Types should not be ignored

Editing Methods and Triggers In the OMS Pro Type Editor, the user can open another window for adding and editing both methods and triggers. This window is opened from

within the type's pop-up menu. Unfortunately the user can be sure that both the method and trigger editor window are not well shaped. Apart from this poor visual representation, the OMS Pro Type Editor disappears from the display and can only be reopened from within the Windows system task bar at the bottom of the screen.

Type Attributes As we have seen from the 'Strengths' section, the attribute details are conveniently presented according to their DDL declaration. The actual weakness however is that the OMS Pro Type Editor implements the functionality for adding, removing and editing attributes in an unsatisfactory way. We discuss these weaknesses within the following three items.

- Adding Attributes

  For adding new attributes, the user first of all has to put the editor in the 'open attributes mode'. Then he can click on a type's name and choose the entry 'New' from the pop-up menu. After that, the user can enter the new attribute's name and its type. However, the type cannot be browsed for and the user can only hope for it to exist in the database. When there is a bulk property to be specified, the user must type in the bulk property value followed by the string ' of ' in exactly that manner and is not forgiven any typing errors. Last but not least, whenever an attributes list exceeds the number of five, the list is cut in its visual representation. This may contribute to a clear object arrangement even in the 'open attributes mode', but is not intuitive and at first very confusing to the user.

- Editing Attributes

  When a user tries to edit attributes, he cannot just press the return button for saving the changes. He has to click on an attribute's name and choose the entry 'Save'. Instead of saving the changes immediately, the user is asked within a dialog if he wants to alter the attribute or to create a new one. From our point of view, this dialog is redundant. If the user intended to create a new attribute, he would have chosen the 'New' entry from the type's pop-up menu.

- Deleting Attributes

  Deleting attributes is a side effect of editing them. Instead of offering a 'Delete' entry in a pop-up menu, an attribute is deleted by removing all letters of its name. Again this procedure is not intuitve.

Last but not least, the OMS Pro Schema Editor does not provide a functionality to change the order of attributes.

2.4 OMS Pro Classification Editor

2.4.1 Strengths

Adding Constraints The procedure to add new constraints is very intuitive. By clicking on the 'Add Constraints' icon in the tool bar of the OMS Pro Classification Editor, the user is able to choose all the relations required for the constraint to be built. Upon a specific

selection, the pop-up menu does only contain the set of entries that are actually allowed to choose. Therefore the risk of database inconsistency is eliminated already at its earliest stage, i.e. on the graphical level.

Editing Collection Names Collection names in the OMS Pro Classification Editor are altered by clicking on the context menu entry 'Change Properties' or by clicking on the icon 'Edit Names' from the tool bar. The second alternative gives the user the opportunity to *directly* edit the collection names. An even more elegant solution to alter a collection's name would be a functionality for directly editing the names *without* having to put the editor into a special 'name editing mode' beforehand.

2.4.2 Weaknesses

Collection Names When dealing with the default preferences, collection names longer than ten letters exceed the border of a collection. The user gets the alternative to change the default preferences on the 'Preferences Page' within the 'Schema Editor' section. Preferable however would be an automatic layout, i.e. an adaptation of the font size within the individual collections dynamically according to the length of their names. Another alternative would be to automatically adapt the size of a collection's shape.

Adding a Collection For adding a collection in the OMS Pro Classification Editor, the user opens the 'New Collection' dialog window where he enters the collection's name and its type. However, the type cannot be chosen from a list, i.e. the type of a collection cannot be browsed for. The user has no other alternative than to go back to the main application window and to look up the type's name manually. Since the OMS Pro Classification Editor does not validate the type name at run time, this procedure is quite error prone. It rather tries to add the collection to the OMS database in either case and throws an error on failure.

Sub and Super Collections Right clicking on a specific collection opens a context menu from where the entries 'Be Supercoll of' or 'Be Subcoll of' can be chosen. Drawing the connections directly into the diagram would be more comfortable.
Furthermore, just as within the OMS Pro Type Editor (see Sect. 2.3.2), indirect super and sub collection should not be ignored.

Associations Associations are added by clicking on the 'Edit Associations' icon and then using drag and drop from collection to collection. Unfortunately, upon such an event there is neither a dialog asking the user to give a name to the association, nor there is a dialog to enter cardinality values for both source and target collections. The name of the association is simply created from the concatenation of the collection names, and the cardinality is set to (0,*) by default.

Changing an Association's Cardinality Changing the cardinality is possible by double clicking on the connection lines. In the association object window, new values for the source and target cardinality can be entered and the database model is updated. Yet the OMS Pro Classification Editor gets no update notification, not even when the association object is closed.

Changing an Association's Name There is the possibility, apart from switching to the 'edit text mode', to change an assocation's name by selecting the 'Change Properties' entry from its context menu. Astonishingly, this leads to the deletion of the assocation's source and target connection lines.

Constraints In addition to the advantages described in the 'Strengths' section, there is also a bad characteristic that concerns the creation of constraints. Once having clicked on the 'Add Constraints' icon, the decision for a specific relation is irrevocable. In other words, there is no way to undo a selection, neither by clicking anywhere in the editor area nor by clicking on the relation again. The user is given no other choice than to restart the procedure, i.e. to select the arrow icon and to choose the constraint button again.

3 Graphical Editing Framework

We knew right from the start that there are the standard Java tools to build Graphical User Interfaces (GUI) and that they are provided by the Swing, AWT and SWT libraries. Since the whole Eclipse GUI is based on the `org.eclipse.swt` package, it has never been under consideration to work with any library other than the SWT toolkit. The main part of this package is represented by `org.eclipse.swt.widgets`. It essentially contains the implementation of graphical objects.
However, the SWT library alone turned out to not suit our requirements entirely. The application must provide functionality for visualising a tree structure with the possibility to connect nodes on different hierarchy levels. In addition, it must offer a printing mechanism and provide the possibility to save and load layouts. Eclipse has a solution to fulfil many of these requirements, the Graphical Editing Framework (GEF).
GEF is a powerful foundation to create editors that represent arbitrary models visually. The disadvantage of this generic framework, like the disadvantage of so many generic frameworks, is that its comprehensive design makes it difficult to apply. However, once learned, GEF becomes an invaluable tool particularly when dealing with entities that need to be connected. A detailed description of the core GEF concepts is found in [4] and [8].

3.1 Core GEF concepts

GEF is based on the model-view-controller paradigm. In general, the model describes the data, the view represents the model graphically, and last but not least the controller is responsible for handling user input, making changes on the model if necessary, and refreshing the view. In the following, we give a quick overview of these basic elements.

3.1.1 Model, View and Controller

Model The GEF developers designed a framework that works with any data. The model should not know anything about the controller or the view.

View The view is the visual representation of the model. GEF normally uses figures from the `org.eclipse.draw2d` package. The view should remain ignorant of both the model and the controller.

Controller In GEF terminology, the controller is also called edit part. It represents the component that brings the view and the model together.

Building controllers At the beginning, a top level controller is created which corresponds to the top level model. If the model represents a hierarchy, the top level controller allows child controllers to be created for each element in the model hierarchy tree. This procedure is done recursively until all hierarchy levels have their controllers built. These controllers, acting as a mediator between model and view, are able to build the view according to the contents of the model.

Notification Since neither the model nor the view know about each other, it is the task of the controller to listen to changes in the model and to update its visual representation. The most common pattern used for GEF is a model that posts notification messages to the controllers. The edit parts then react appropriately by adjusting the model's visual representation.

3.1.2 User interaction

GEF offers a so-called 'palette'. From the palette, the user can select appropriate tools for editing the schema diagram. Such a tool translates low level events into a high level requests, represented by request objects. In the majority of cases, tools post their requests to the controller whose figure was underneath the mouse arrow when the mouse button was pressed. Sometimes the request is also posted to various controllers. The `MarqueeSelectionTool` for example posts its request to all edit parts whose figures were contained within the selected area. Regardless of how many controllers are chosen as the target of a request, the controllers do not handle the requests themselves. Each controller class registers so called edit policies beforehand so that the controllers are able to outsource the request handling.
A policy not wanting to handle a specific request may return a null reference. However, this is a rather inelegant style of programming. It is not supposed to be a policy's task to decide about the request being handled or not. Rather it is the controller's task. Hence an edit part that does not want to react on specific requests simply doesn't install the appropriate policy. The mechanism of keeping the controllers and the policies separated allows the reuse of policies for multiple controller classes. Both controllers and policies are specialised for their tasks and work independently from each other.

3.2 Extending GEF

For visualising an OMS database, relying on the exisiting GEF functionality is not enough. For instance, we need to create connections between multiple objects.

3.2.1 Connecting Multiple Anchors

GEF supports drawing connections between two objects. For the visualisation of the cover, disjoint, partition and intersection constraints however, the need for multiple anchor connections arises. One possible solution would have been to implement real multiple anchor support and thereby modifying built-in GEF classes on a high abstraction level. The main disadvantage, apart from resulting in a huge programming effort, is that the resulting behaviour is not exactly what we expect to have. What we do want to have for drawing a constraint arc is a way to always connect the two outermost connection lines along the x-axis. We finally had the idea of creating 'pseudo multiple anchors'.

3.2.2 Pseudo Multiple Anchor Connections

We searched for a way to always keep the two outermost connection lines of a constraint up-to-date. The most reasonable approach to determine the outermost connection lines is to consider the position of the point where a connection crosses the border of the object along the x-axis. But we will see in this section that this approach causes, upon moving collections, problems concerning the identification of the outermost relations.
When we *create* new constraints, we get the coordinates of the crossing points easily. The layout manager has already laid out the diagram, the connections are established and the crossing points can be calculated.

Determining Dynamic Anchors: Problems In contrast, the connections are not yet established when we move a collection in the diagram. The collection bounds are already set, but the connections between the entities are not established until GEF has laid out the diagram. Therefore there is no easy way to get the outermost relations and update the constraint anchors beforehand.
Changing the model is neither possible after GEF has laid out the diagram. The `layout` method assumes that the models and controllers are established properly. Changing the source or target anchor of a constraint however means altering the corresponding relationship models.

Determining Dynamic Anchors: Solution Nevertheless, we have found an answer to this problem. Our solution is to take a collection's position along the x-axis on the opposite side of the relation as reference. Although there are situations which are visually inferior to taking the current crossing points, in practice this turned out to be a thoroughly satisfactory solution. The constraint model itself has no other functionality than being responsible for adding a reference of itself to the source and target anchor objects. The following two items describe the procedure of creating multiple anchor constraints and the procedure of updating the constraint anchors upon moving objects.

- Creating Constraints: When we create constraints within the OMS^{jp} Schema Editor, we get the positions of the collections associated to the involved relations. We add a reference of the constraint to the relation model instances belonging to the two outermost collections.
- Moving Objects: The rearrangement of the anchors comprises the deletion of an existing connection and the reestablishment of a new one. When moving objects, we first

check all the current constraint model instances. We have access to all these instances since we give a reference of the topmost user controller to the command that is responsible for moving objects. The topmost user controller knows about all the constraints.

For all these constraint instances, we remove the references to themselves from their current source and target relations. Then we search for the currently outermost collections, look for their relations and add a reference of the constraint model instance to these source and target relation model instances.

This procedure could further be improved by only inspecting and rearranging the constraint model instances that are actually involved.

3.3 Schema Editor Design

In this section we give an overview of the OMS^{jp} Schema Editor design and the problems that have occured. The section can be seen as an introduction to Chap. 4. In chapter 4 we give a detailed analysis and description of the application's implementation.

3.3.1 Creating a Custom Decorator Model

The OMS^{jp} Plugin has transformed the OMS Object Model into a tree structure model. Since this approach in principle also fits the GEF conventions, we have adopted this tree structure for the OMS^{jp} Schema Editor.
The functionality of the various objects however had to be extended for multiple reasons. First of all, it was indispensable to provide a notification mechanism. In addition, supplementary classes for representing relationships had to be introduced. The GEF convention prescribes that each connection is indicated by an separate instance of a relationship model class.
Extending the OMS^{jp} Plugin classes would have been a possible approach for building the OMS^{jp} Schema Editor model. But, since all these classes must provide a general mechanism for adding listeners, a large number of independent extensions is impractical. The most flexible approach is to let new objects enclose the elements from the tree structure. For this reason, we introduced the so-called decorator pattern. It allows responsibilites to be added and removed from the OMS^{jp} Plugin model classes at run time. Moreover, this approach preserves best one of the fundamental rules in the object-oriented design, Separation.

3.3.2 Problems Concerning the Decorator Model

Unfortunately this design is also the basis for new problems. We describe them in the following paragraphs.

Relations The tree structure is wrapped into the OMS^{jp} Schema Editor model when the `Schema` class is instantiated. A reference to the OMS^{jp} Plugin root model is given to the `Schema` in its constructor and the decorating model instances are created recursively until the entire tree is generated.
A problem occurs when we try to add relations among the types or collections of our enclosing data structure. When we simply follow the algorithm described above, we end up losing the information of how our types and collections are related to each other. Assuming we want to establish the relations after we have created all our nodes in the tree. We can find all direct

super nodes by visiting the decorated object, but once they are found we have no reference back to our enclosing instances. However, there is a solution, even though an inefficient one. To find the enclosing instance, we have to go through all our nodes on a particular level and to ask each of them if it is referencing the wrapped object.

Notification The OMS^{jp} Schema Editor has to be informed of the changes in the OMS^{jp} tree structure, but the OMS^{jp} Plugin model has no knowledge of all its enclosing decorators. We therefore had to search for a way to update the data in the OMS^{jp} Schema Editor when required. We considered the idea of introducing a thread that updates the data in certain intervals. But the disadvantages are obvious:

- The data and its view get not perfectly synchronised. This is confusing for the user and might lead to undefined states.
- Threads are a performance gap.
- Last but nod least, it is inelegant programming style to catch events by continuous procedures.

The most natural approach therefore is to add the OMS^{jp} Schema Editor as a listener for event notifications to the underlying OMS^{jp} Plugin. This ensures that upon any change in the OMS^{jp} Plugin model, the OMS^{jp} Schema Editor gets informed of the model's current state. For a detailed description of the notification mechanism, the reader is referred to Sect. 4.4.

4
Schema Editor Implementation

The classes of the OMS^{jp} Schema Editor are identified by the string `ch.ethz.globis.omsjp.omsbrowser.schema`, and they are arranged in packages. The application is divided into three parts. The packages in these three parts contain classes for the Classification Editor (denoted by the extension `.classification`), for the Type Editor (denoted by `.type`), and for both (no extension).
Nevertheless, we have decided to describe the implementation details *not* ordered by packages. We prefer giving a description of how the classes work together instead of describing them and their functions isolated within their packages. Since the logical control flow is not ordered by packages, we think that this approach is more useful for a programmer who wants to get familiar with the details of the OMS^{jp} Schema Editor implementation.
From time to time, we intentionally violate conventions in the sense of leaving out unnecessary information that does not effect a better understanding. The figures in this chapter sometimes may not strictly follow the UML conventions. For instance we decided to use the normal sub class indication lines all over, even though a class might only be an *indirect* sub class. We do this to keep the diagrams simple and comprehensible, cases like the one described above are explained in the text. After all, the applied symbols and rules are used consistently throughout the chapter.
The table below explains the short cuts used to indicate the visibility of classes, methods and variables within the diagrams. These symbols accurately follow the conventions.

SHORT CUT	VISIBILITY
'+'	public
'#'	protected
'-'	private

The diagrams in this chapter can be seen as a way to illustrate the text. Vice versa, the text can be seen as a way to explain the diagrams. The best way to read this chapter is to combine its textual and visual representation. We tried to support all textual passages by methods and

implementation fragments in the diagrams. However, occasionally there may still occur a textual passage that is not illustrated by a diagram's class or method.
In the general overview we have seen that a GEF application consists of three parts. In Sect. 4.2 and Sect. 4.3 we will show how the view and the controller parts are implemented concretely. Since the model is referred to in the controller part, we have decided to leave out a separate section for its description. In the following sections of this chapter we will illustrate how the notification mechanism works, how the OMS^{jp} Schema Editor interacts with the OMS^{jp} Plugin and how the OMS^{jp} Schema Editor deals with the users' interaction.

4.1 Building a 'Plugin for a Plugin'

The chosen model design inspired us to build an independent OMS^{jp} Schema Editor Plugin. This OMS^{jp} Schema Editor Plugin adds the OMS^{jp} Plugin to the requirements list in the dependency section of the file `plugin.xml`, as depicted in Fig.4.1.

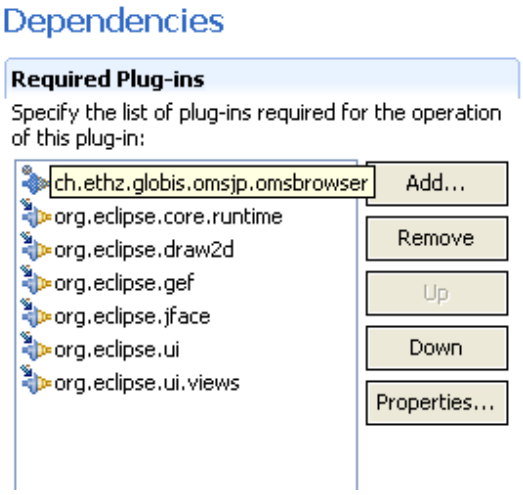

Figure 4.1: Adding the OMS^{jp} Plugin

4.2 View

First we were thinking about integrating the type and classification editors within a so-called multipage editor. But as the editors work independently of each other for the most part, there is no compelling reason to do so. We therefore decided to keep all the editors separated and created the classes `UserTypeEditor`, `SystemTypeEditor`, `UserClassificationEditor` and `SystemClassificationEditor`. All of them are indirect sub classes of `GraphicalEditor`. Since the user and system editors do not significantly differ in their conceptual desing, throughout this chapter we describe the implementation procedure examplarily by means of the user editors only.

4.2.1 Initialising `ScrollingGraphicalViewer`

GraphicalEditor by default creates an instance of `ScrollingGraphicalViewer` in its protected method `createGraphicalViewer`. However, we have to initialise additional settings like the contents of the viewer when the editor gets started. We do this best by overriding the `createGraphicalViewer` method and creating our own graphical viewer, an instance of `ScrollingGraphicalViewer`. The viewer is created by means of the `GraphicalViewerCreator` class and can be accessed through the `getGraphical-Viewer` method.

4.2.2 Configuring the `ScrollingGraphicalViewer`

It is highly important to configure the `ScrollingGraphicalViewer` properly as it plays an essential role when displaying the diagrams. There are mainly two issues to consider. They are described in the following two paragraphs, and Fig. 4.2 shows the class diagram.

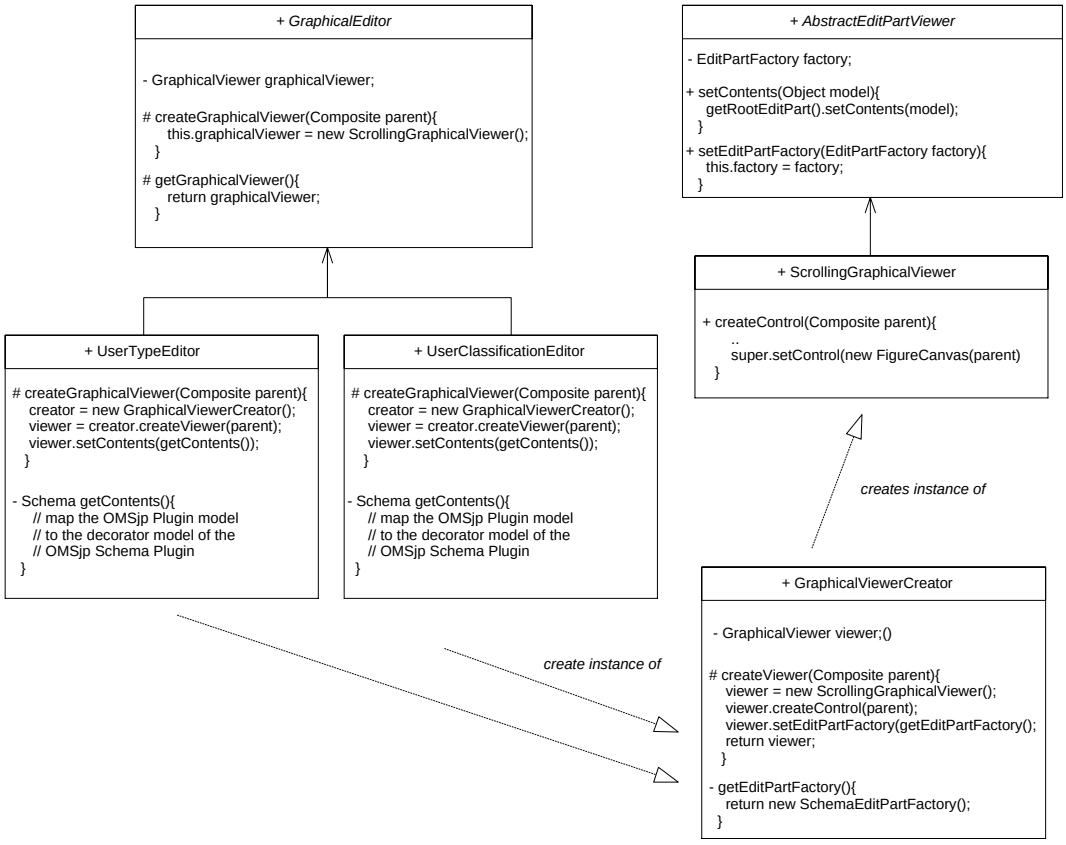

Figure 4.2: Configuring the View

Setting the Factory The `GraphicalViewerCreator` configures its viewer with an edit part factory. `ScrollingGraphicalViewer` is a sub class of `Abstract-EditPartViewer`, and this class in turn provides the method `setEditPart-`

`Factory`. Therefore `GraphicalViewerCreator` simply calls `viewer.setEditPartFactory(factory)` within its method `createViewer`. We will see in the next paragraph that the edit part factory is indispensable for establishing a link between the model and its view.

Setting the Model Furthermore, the diagram's model has to be passed to the viewer. `AbstractEditPartViewer` provides a method `setContents(Object contents)`, and all we do to set the viewer's contents is calling `getGraphicalViewer().setContents(model)` from within the `createGraphicalViewer` method in the user type editor and the user classification editor.
The information of the model is needed by the `AbstractEditPartViewer` to create the root edit part. `AbstractEditPartViewer` creates the root edit part by calling `getEditPartFactory().createEditPart(null, model)` in its `setContents` method.

4.3 Controller

The `AbstractEditPartViewer` is not only responsible for setting the model and the factory, it also lays the foundation for the contents to be *displayed*. By means of the controller factory and the model, `AbstractEditPartViewer` ensures the proper construction of all the user edit parts. We describe this procedure in the following section.

4.3.1 Creating Controllers

First of all, an instance of `SchemaEditPart` is constructed. With the reference to the model and to the controller factory, the `AbstractEditPartViewer` ensures that this instance is created within the factory's `createEditPart` method. The `SchemaEditPart` directly follows the root edit part in the hierarchy and controls the schema diagram. Whenever the underlying model of a controller has any children, the edit part ensures to provide a method for accessing them. This method is named `getModelChildren`. The `EditPartFactory` recursively creates controller instances for these children until an edit part only returns the empty list in its `getModelChildren` method.
In this manner GEF establishes a one-to-one mapping from the model instance to the edit part, and a one-to-one mapping from each edit part to its figure. Each controller is aware of its model and maps information from this model to a graphical representation, which is accessed in the controller's `createFigure` method. The diagram depicted in Fig. 4.3 illustrates the general scheme of this mapping procedure. Note that, for visualising the tree structure, the edit parts on the right hand side do not represent classes but *instances* of classes.

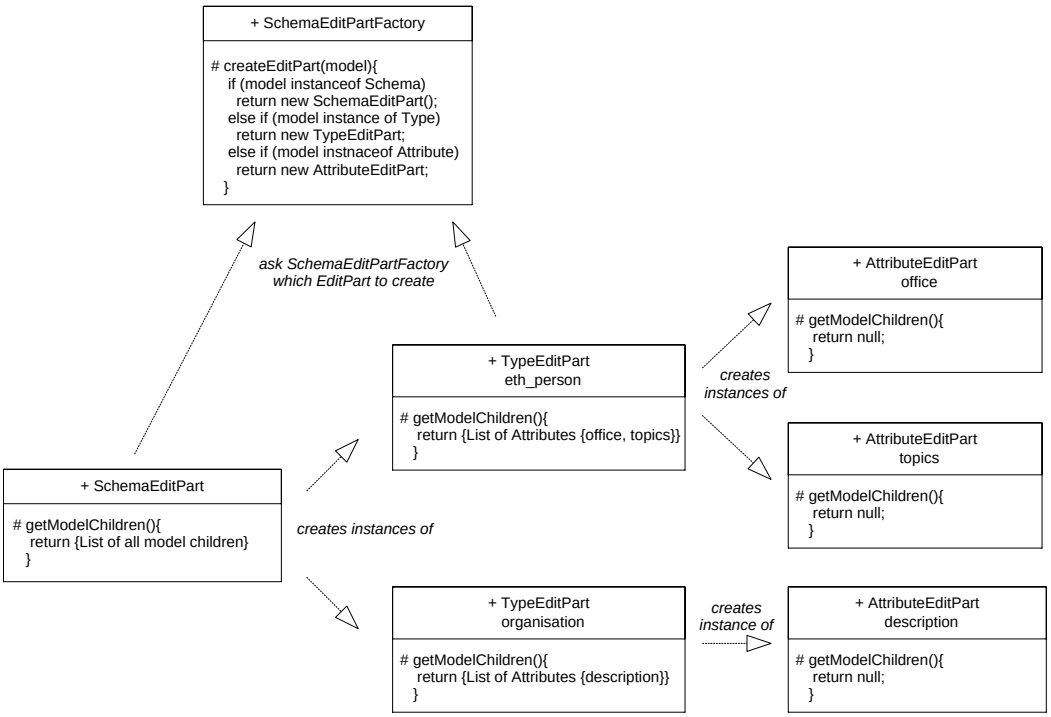

Figure 4.3: Controllers are instantiated according to the Model

4.3.2 Establishing Connections

In addition to the `getModelChildren` method, a controller optionally provides the two methods `getModelSourceConnections` and `getModelTargetConnections`. Both methods return a list of connection model objects, they can be seen in the class `ObjectEditPart` of Fig. 4.4.
These methods serve as indication of a connection's start and end point. Each connection model instance is referenced exactly twice. One reference is in the list of source objects in the model instance from where the connection starts, and the other reference is in the list of target objects in the model instance where the connection ends.

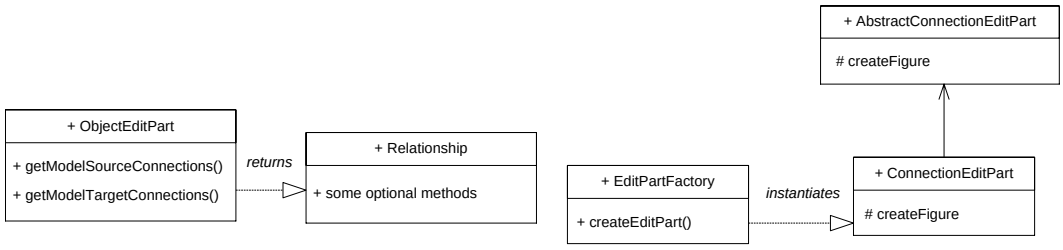

Figure 4.4: Creating Connections

GEF ensures that the factory assigns the connection model object to its corresponding connec-

tion controller. Unlike a 'normal' edit part, such a `ConnectionEditPart` normally does not need to refer to its underlying model for creating the figure. A `ConnectionEditPart` provides references to its source and target controllers and since these controllers in turn provide references to the source and target figures, this is enough information for drawing a line connecting the two objects.

4.3.3 Anchors

It is not obvious where to connect two objects exactly. The position of the connection point is determined by so called anchors. For other classes to have access to these anchors, the edit parts implement the interface `NodeEditPart` and implement the methods `getSource-ConnectionAnchor` and `getTargetConnectionAnchor`. These methods return the anchor point of all relations for which the current object is the target or source respectively. Both methods return a `ConnectionAnchor`, i.e. they return an instance of a class that implements the interface `ConnectionAnchor`.
In the majority of cases these classes do not directly implement the `ConnectionAnchor`, but they subclass `AbstractConnectionAnchor`. This class adds supplementary functionality, one of which is the maintenance of the reference to the figure. The procedure of instantiating anchors is depicted in Fig. 4.5.
The implementation of the OMS^{jp} Type and Classification Editors requires different `ConnectionAnchors`. We describe them in the following.

- `ChopboxAnchor` and `EllipseAnchor`

  Both source and target anchors within a type edit part or a collection edit part are so called chopbox anchors. These anchors are provided by the Graphical Editing Framework, hence at this point we do not discuss them further.

- `RelationAnchor`

  A `RelationAnchor` is required to determine the start and end point of a constraint arc. This anchor must only be used within controllers that provide a `Polyline` as graphical representation of their model.

  The two methods `getReferencePointUpperCircle` and `getReference-PointLowerCircle` are able, depending on the ratio $\frac{a}{b}$, to return any particular point on the polyline. The variables a and b can have arbitrary positive values except from *both* being zero. The methods add up start and end coordinates, weighted by the variables a and b, and divide them by the sum $a + b$. The formal calculation is given below. Applying this formula determines the anchor at distance $\frac{a}{a+b}$ multiplied by the total length from the starting point.

$$x_{new} = \frac{b \cdot x_{start} + a \cdot x_{end}}{a + b}$$
$$y_{new} = \frac{b \cdot y_{start} + a \cdot y_{end}}{a + b}$$

  We tested several values for a and b. However, just taking the endpoint for a 'lower arc' and the startpoint for an 'upper arc' results already in a good visualisation. In the current release, a `RelationAnchor` takes the parameters $a = 0$, $b = 1$ for calculating the start point and the pair $a = 1$, $b = 0$ for calculating the end point.

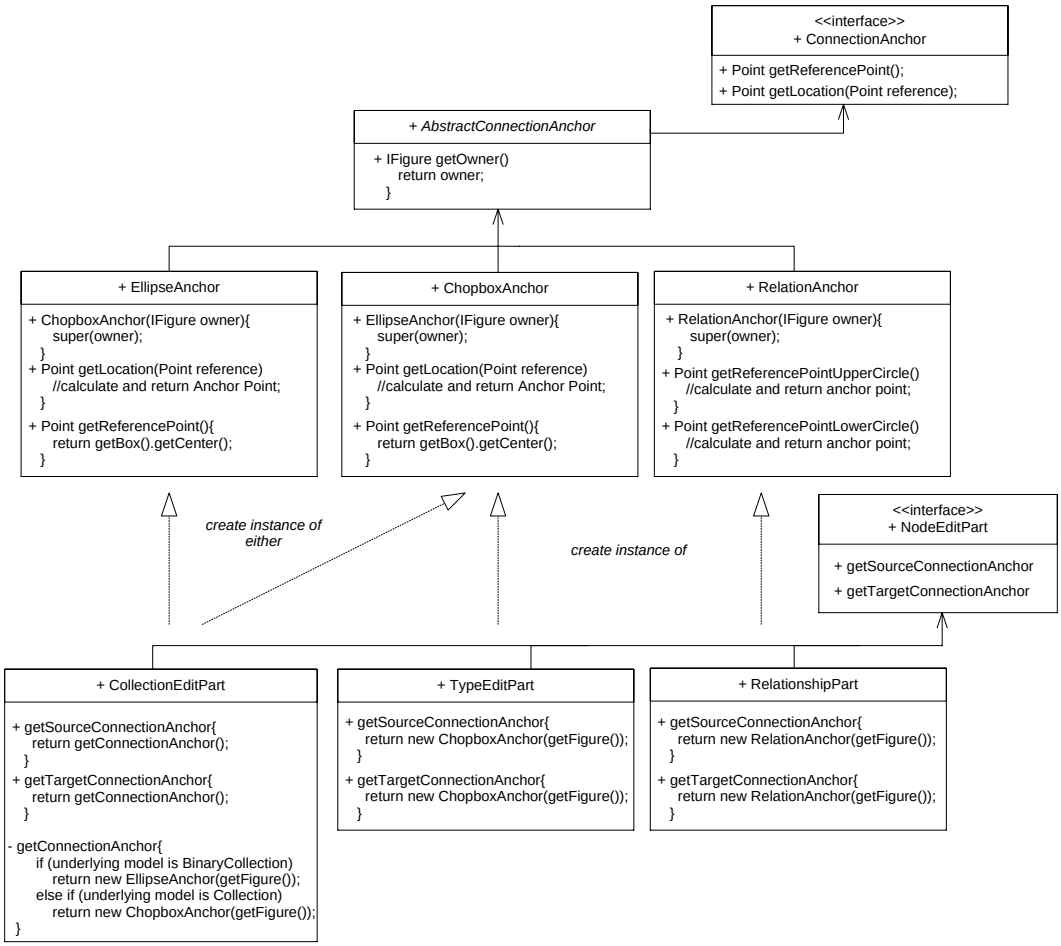

Figure 4.5: Anchors and from where they are instantiated

All the connection figures implement the interface `Connection`. `Connection` is an interface that ensures the implementing classes to provide the references to the source and target anchors. By default, the connection figures are represented by the built-in `Polyline-Connection`.
In its layout method, a connection figure calls the `route` method on an instance that implements `ConnectionRouter`. The `PolylineConnection` for example calls this method on an instance of the class `NullConnectionRouter`. This mechanism can be seen in Fig. 4.6.
However, the constraints within the OMS^{jp} Schema Editor require a different kind of connection line. Constraints do not connect objects with a straight line, but with an arc. The connection figure classes `ArcConnectionDown` or `ArcConnectionUp` therefore call the `route` method on instances of the classes `ArcConnectionRouterLowerHalf` or `ArcConnectionRouterUpperHalf` respectively.
The task of the `route` method is the setting of a list of points for the connection. The final connection line is a line connecting all these entries from the list. In the following we describe the different `route` methods in detail.

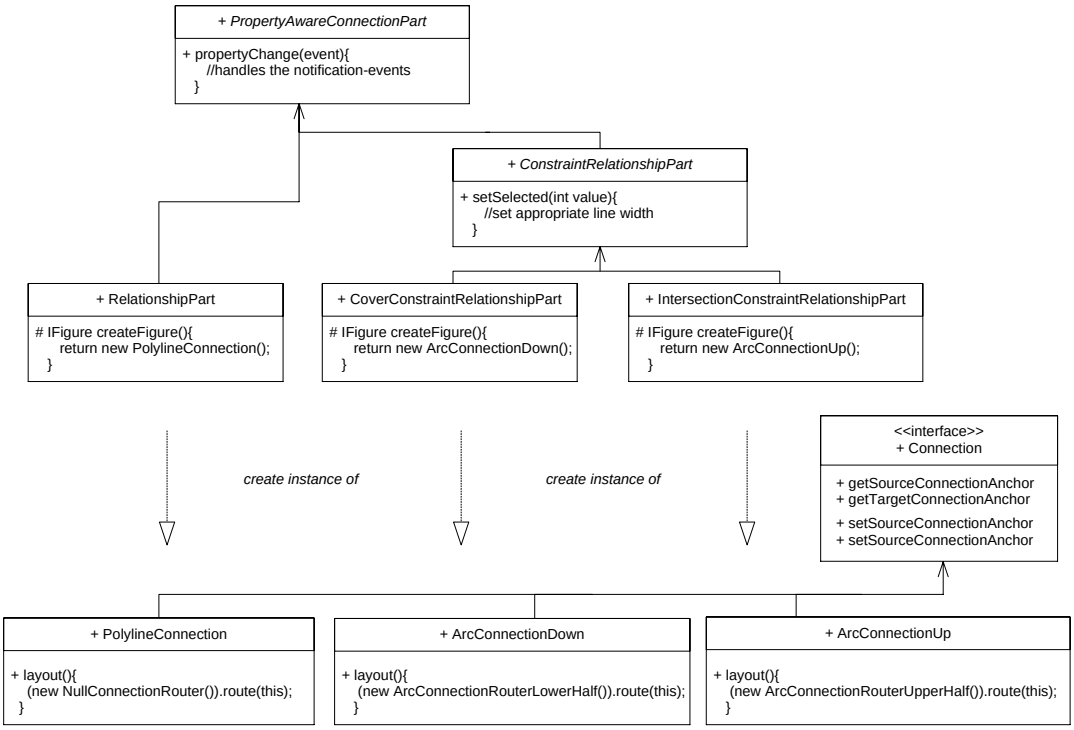

Figure 4.6: Different Connections create different Routers

`NullConnectionRouter` The only thing a `NullConnectionRouter` does within its `route` method is to add the source and target point to the point list of the connection. `NullConnectionRouter` has got their references directly from the source and target anchors in the connection figure.

`ArcConnectionRouterLowerHalf` Within the `route` method of this class we basically need to calculate the three values radius, midpoint and angle.
The basis for the calculation of these values are the points from the `getReference-PointLowerCircle` method in the relations' source and target relation anchors (see Fig. 4.5).

- Midpoint

  The midpoint is calculated by calling the `getReferencePointLowerCircle` method within both the source and target relation anchors and taking their x coordinate. Adding up the two values and dividing them by 2 brings out the midpoint in the horizontal direction. We do the same for the y coordinate, as it can be seen from the following two formulas.

$$x_{midpoint} = \frac{x_{sourceReferencePoint} + x_{targetReferencePoint}}{2}$$
$$y_{midpoint} = \frac{y_{sourceReferencePoint} + y_{targetReferencePoint}}{2}$$

- Radius

  The radius is calculated by taking the square root of the sum from the squared distances between the midpoint and one of the two relation anchors in the x and y direction.

$$r = \sqrt{(x_{midpoint} - x_{sourceReferencePoint})^2 + (y_{midpoint} - y_{sourceReferencePoint})^2}$$

- Angle

  The angle is calculated according to the following formula:

$$\alpha_{rotate} = \arctan(\frac{x_{targetReferencePoint} - x_{sourceReferencePoint}}{y_{targetReferencePoint} - y_{sourceReferencePoint}})$$

Now the arc is drawn with an i $\in [\alpha_{rotate}, \ldots, \alpha_{rotate} + \Pi]$ and a default increment of 0.01.

$$x_{arc} = r \cdot cos(i)$$
$$y_{arc} = r \cdot sin(i)$$

After all, the points have to be translated to the midpoint:

$$x_{arc_{translated}} = x_{arc} + x_{midpoint}$$
$$y_{arc_{translated}} = y_{arc} + y_{midpoint}$$

Finally, all the calculated points $(x_{arc_{translated}}, y_{arc_{translated}})$ are added to the point list.

`ArcConnectionRouterUpperHalf` This class is needed for the visualisation of the intersection constraints. The arc is calculated similarly, but there are two differences. First, instead of calling `getReferencePointLowerCircle` from the `RelationAnchor`, the method `getReferencePointUpperCircle` is called. Secondly, the iteration for getting the points on the arc is done for an i $\in [\alpha_{rotate} + \Pi, \ldots, \alpha_{rotate} + 2 \cdot \Pi]$ with the same increment.

4.4 Notification mechanism

It is important to be aware of the fact that the notification does not originate in the OMS^{jp} Schema Editor itself. Creating and editing objects is an integrated part within the OMS^{jp} Plugin, and since the OMS^{jp} Schema Editor builds its model by 'decorating' the model classes of the OMS^{jp} Plugin, it does not work independently. Therefore we had to find a way to be informed of the changes in the OMS^{jp} Plugin data structure.
As already mentioned in Sect. 3.3.1, we first considered taking the OMS^{jp} Plugin model as a basis for our implementation and integrating the GEF notification mechanism therein. We further know from that chapter that this would have caused severe difficulties, in particular since the OMS^{jp} Schema Editor introduces new data structures such as relationships. The OMS^{jp} Plugin itself would have been responsible for keeping these supplementary classes up-to-date. This would have been contrary to the idea of separating classes into logical components, thus we had to find another solution.

After all, the only solution was to build the OMS^{jp} Schema Editor as a separate logical unit by giving the OMS^{jp} Schema Editor its own model. We have seen that there is an one-to-one correspondance among the model, controller and the view in the Graphical Editing Framework. Hence, each object in the OMS^{jp} Plugin had to be given its own decorator class in the OMS^{jp} Schema Editor. The sole exception is the relationship which has no correspondance in the OMS^{jp} Plugin and therefore does not represent a wrapper class. Nevertheless, all these classes are prepared to get decorated with new functionalities like the GEF notification mechanism.
The notification mechanism can be divided into two parts. First, the OMS^{jp} Schema Editor model gets notified about any modification in the OMS^{jp} model. Then, the GEF internal notification mechanism propagates the changes to its view.
In the following we illustrate this two-way notification mechanism, starting with the GEF internal notification service. Since GEF is model neutral, our application has to add its own listeners to handle the notifications.

4.4.1 GEF Internal Notification Mechanism

Whenever an `EditPart` is created, GEF ensures that the method `activate` from `AbstractGraphicalEditPart` is called. We override this method in the class `PropertyAwarePart` to be sure that the corresponding edit part is added as a listener to its underlying model. Contrary, when a controller is not required anymore, it is important to remove it from the list of listeners in the class `PropertyAwareObject`. We therefore override `deactivate`. If we did not, the system would very soon suffer from lack of memory. The diagram in Fig. 4.7 shows this procedure and the mechanism desribed in the following two paragraphs.

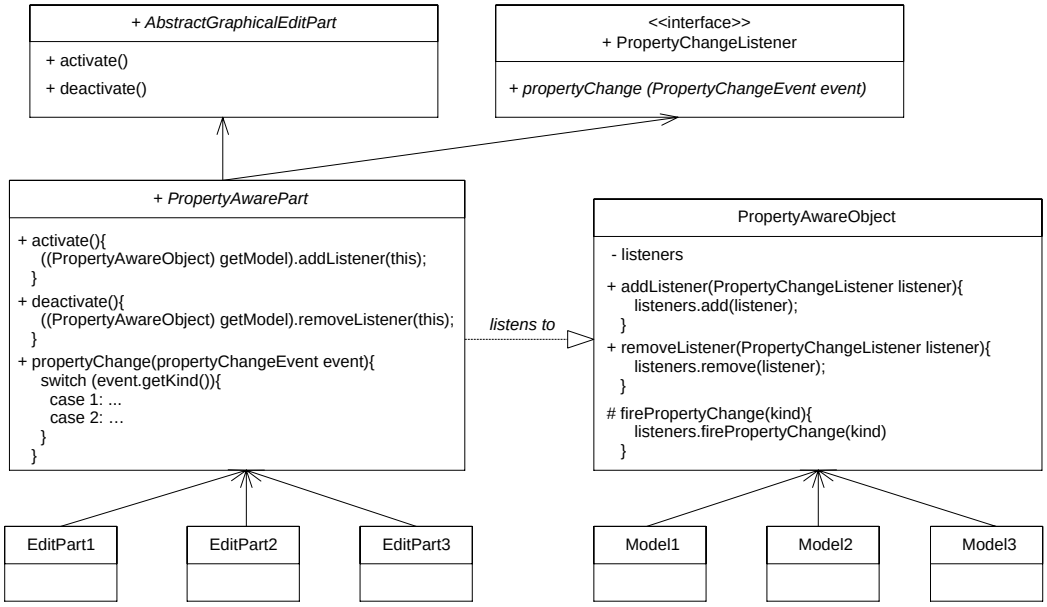

Figure 4.7: Notification Mechanism on the Schema

PropertyAwarePart Instead of extending `AbstractGraphicalEditPart` directly, each edit part subclasses `PropertyAwarePart`. As it is described above, this class adds and removes instances of itself in the corresponding models. For listening to changes, it overrides the abstract method `propertyChange` from the interface `PropertyChangeListener`. Within this method it decides for performing the proper udpates upon a specific property change event.

PropertyAwareObject The models on the other hand are sub classes of `PropertyAwareObject`. This class offers the possibility to add and remove the property change listeners and provides the method `firePropertyChange` to inform all the listeners of the kind of event and, implicitly, of the objects involved. `PropertyAwareObject` calls `propertyChange` on all its listeners whenever an event occurs. These listeners are actually instances of `PropertyAwarePart`.

4.4.2 Notification from within the OMS^{jp} Plugin

As mentioned earlier, the Graphical Editing Framework depends on notification events from the OMS^{jp} Plugin. The OMS^{jp} Plugin already provides a centralised notification scheme for certain events that ensures objects to be added and removed properly and asserts the accurate modification of object and attribute names.
For a class to be part of this notification mechanism, the OMS^{jp} Plugin provides the method `addListener` in its model class `DatabaseManager`. This method is called with the particular listener instance as a parameter and expects from this instance to implement the interface `IDBElementListener`. Upon an event, the `DatabaseManager` informs all its listeners of this specific event by calling one of the four methods that `IDBElementListener` provides.
Yet this notification mechanism is restricted to a small number of events only. We had to extend the mechanism, thereby maintaining the clear separation between the two plugins. For this reason, we introduced a new interface `IDBSchemaListener` which acts as an extension of `IDBElementListener`.
As depicted in Fig. 4.8, the `NotificationListener` implements this `IDBSchemaListener` and adds in its constructor a reference of itself to the database manager. This enables the OMS^{jp} Schema Editor to listen to all events, in particular also to the events that are of its concern exclusively. The interface `IDBSchemaListener` expects from `SchemaNotificationListener` to implement all the methods required for handling the events. As these methods perform manipulations on the model, the `SchemaNotificationListener` must have a reference to the `Schema` instance. After all, the `SchemaNotificationListener` does have this reference as it is instantiated in the `Schema`'s constructor. In the beginning we thought about letting the Schema class extend the `SchemaNotificationListener`. Passing a reference of the schema however turned out to be the better and more elegant solution as it contributes to the idea of separation. And, last but not least, the model classes are still free to extend other classes if required.
The `SchemaNotificationListener` adopts some of its methods directly from `IDBElementListener`, as for example `dbElementAdded` and `dbElementRemoved`. Other methods however could not be adopted, such as the `dbElementChanged` method. When adding or removing a relation between two objects, the OMS^{jp}

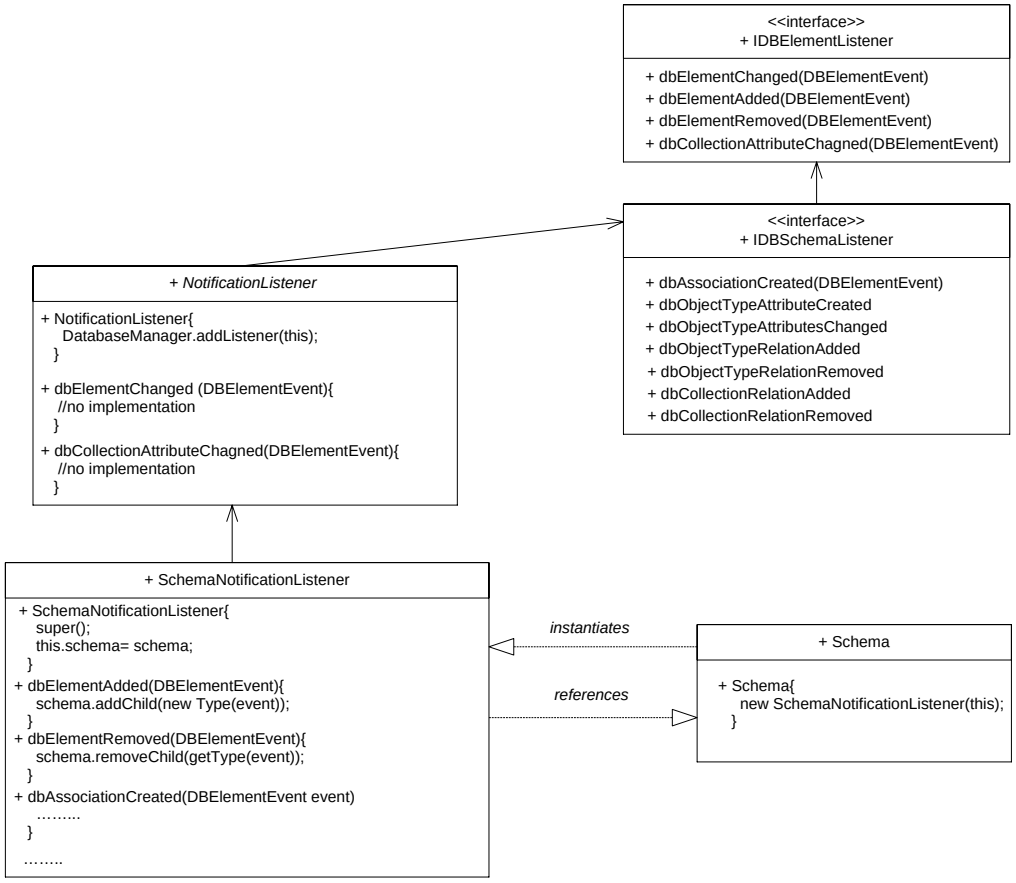

Figure 4.8: Notification Mechanism from the OMS^{jp} Plugin

Plugin fires *two* `dbElementChanged` events for the two objects involved. This may be useful within the OMS^{jp} Plugin environment, but for updating the OMS^{jp} Schema Editor data structure this event handling is fairly unsuitable. We first thought about bringing the two events together by mapping them to a single one. Yet this is risky as we have no idea if the `dbElementChanged` notification is fired in another context too. Hence, we decided to rely on the methods from our `IDBSchemaListener` for handling events that involve modification of a relation.

Removing the Notification Listener when the Editor gets closed When closing either the OMS^{jp} Classification or Type Editor, it is important to remove the `Schema-NotificationListener`'s reference from the database manager. Although Java provides a garbage collector, opening and closing editors without worrying about this issue would make too great demands on the memory.
GEF ensures that the dispose method on `WorkbbenchPart` is called upon closing an editor. Since both `UserClassificationEditor` and `UserTypeEditor` are indirect sub classes of `WorkbenchPart`, we just had to override this method in order to remove the database manager's reference to the `SchemaNotificationListener`.

4.5 User Interaction on the Schema

The two sophisticated notification mechanisms described before simplify the interaction on the schema to a great extent. In most cases, a command simply has to establish the modification on the underlying model of the OMS^{jp} Plugin. Everything else is handled implicitly. The notification mechanism from the OMS^{jp} Plugin fires the update events that the listeners catch, in particular our OMS^{jp} Schema Editor. Then, the GEF internal notification mechanism provides the functionality to update the model's view.

4.5.1 Binding a User Interaction from the Palette to a Command

We show the mechanism on the basis of the OMS^{jp} Type Editor, but binding a user interaction within the OMS^{jp} Classification Editor is done similarly. Figure 4.9 shows the class diagram that illustrates to the following two paragraphs.

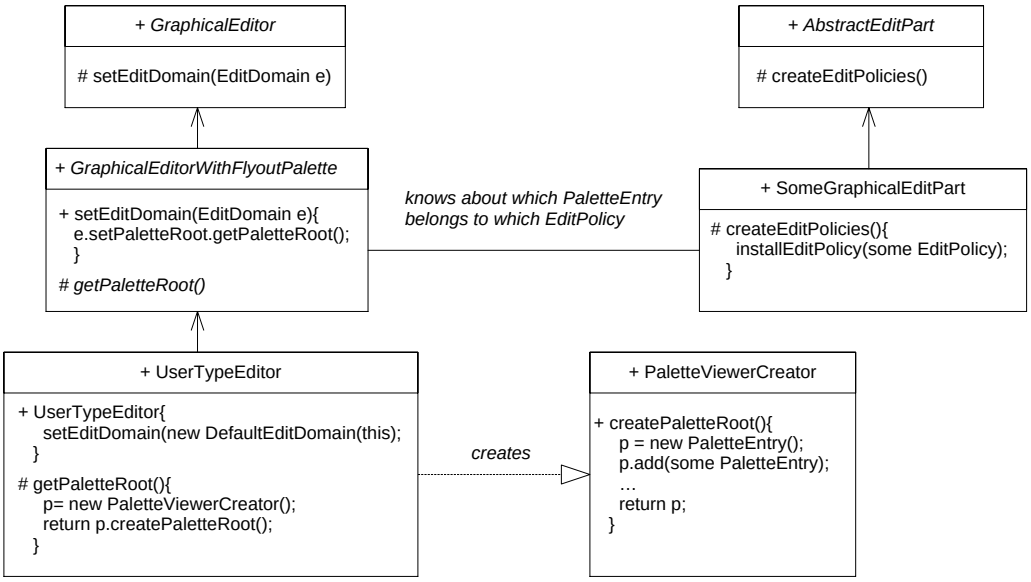

Figure 4.9: Providing a Flyout Palette and Binding the Command

Providing a Flyout Palette Many user interactions require tools from a palette. The `UserTypeEditor` provides such a palette by subclassing `GraphicalEditorWithFlyoutPalette` instead of directly sublassing `GraphicalEditor`. In its constructor, `UserTypeEditor` needs to call the method `setEditDomain` from its super class. This method is responsible for setting the `PaletteRoot`. It calls the abstract method `getPaletteRoot` for which the `UserTypeEditor` offers the concrete implementation. `UserTypeEditor` constructs the `PaletteRoot` by creating a new instance of `PaletteViewerCreator` and calling its `createPaletteRoot` method. Therein, an instance of `PaletteRoot` is created, new `PaletteEntries` are added and finally the `PaletteRoot` is returned. In the following paragraph we describe the way from such a specific palette entry to the execution of a command.

Binding a Palette Entry to an Edit Policy There are different kinds of `PaletteEntries` such as for example the classes `CombinedTemplateCreationEntry` or `ConnectionCreationToolEntry`. GEF establishes a link from the `PaletteEntry` to the corresponding edit policy depending on the particular type of the `PaletteEntry`.
A mouse click within the editor ensures that the `PaletteEntry` calls the `getCreateCommand` method in its assigned policy. This method returns a command that is normally initialised with a reference to the model. By means of this reference the command is able do the proper modifications in the model.

4.5.2 Binding a User Interaction from the Schema to a Command

When the right mouse button is clicked on an object within the editor environment, a pop-op menu appears. The entries therein depend on the current selection of objects. In this section we describe how GEF enables the application to show the proper entries.

First of all, within our editors we override the protected method `createActions` from `GraphicalEditor`. Therein, we register *all* the required actions to the action registry that is provided by the super class `GraphicalEditor`. To be able to register our actions, we save all action ids in a list called `editPartActionIds` by means of the private method `addEditPartAction`. This list of action-ids is located in the user type editor and user classification editors themselves. The action ids identify the actions uniquely. Required actions are, among others, 'delete', 'undo' and 'redo'.
Yet we do *not* want to show all the entries of a pop-up menu at any rate. The `GraphicalEditor` therefore implements the interface `ISelectionListener`. GEF ensures that its method `selectionChanged` is called upon every user interaction. `GraphicalEditor` offers a default implementation for the method `selectionChanged`, but just relying on this implementation causes the entries still to be independent of the actual selection. Hence it is indispensable for us to override the method in both the user type editor and user classification editors.
We have implemented the following mechanism. We override `selectionChanged` in our concrete editors and call the method `updateAction` with our action list as argument. `GraphicalEditor` now 'gathers' all the actions uniquely identified by their action id and calls each action's `update` method, provided in the abstract class `SelectionAction`. This `update` method in turn calls, among other things, each action's run method. Finally the class `WorkbenchPartAction` has a link to the `WorkbenchPart` and therefore is able to refresh the display.
Although the `run` method of *each* action gets called upon a `selectionChanged` call, the system has to know about the *specific* entries of the pop-up menu. An entry only shows up in the pop-up menu when the controller belonging to the selected objects has installed the proper policy. The action's run method is called anyway, but the pop-up menu contains the entry only if the initialised action knows where to redirect the command.
The various actions mainly differ in the implementation of their `run` method. However, they all have in common that the final aim is to get the `commandStack` from the `WorkbenchPart`, provided in `WorkbenchPartAction`, and to execute their specific commands on it. The whole procedure is illustrated in Fig. 4.10.

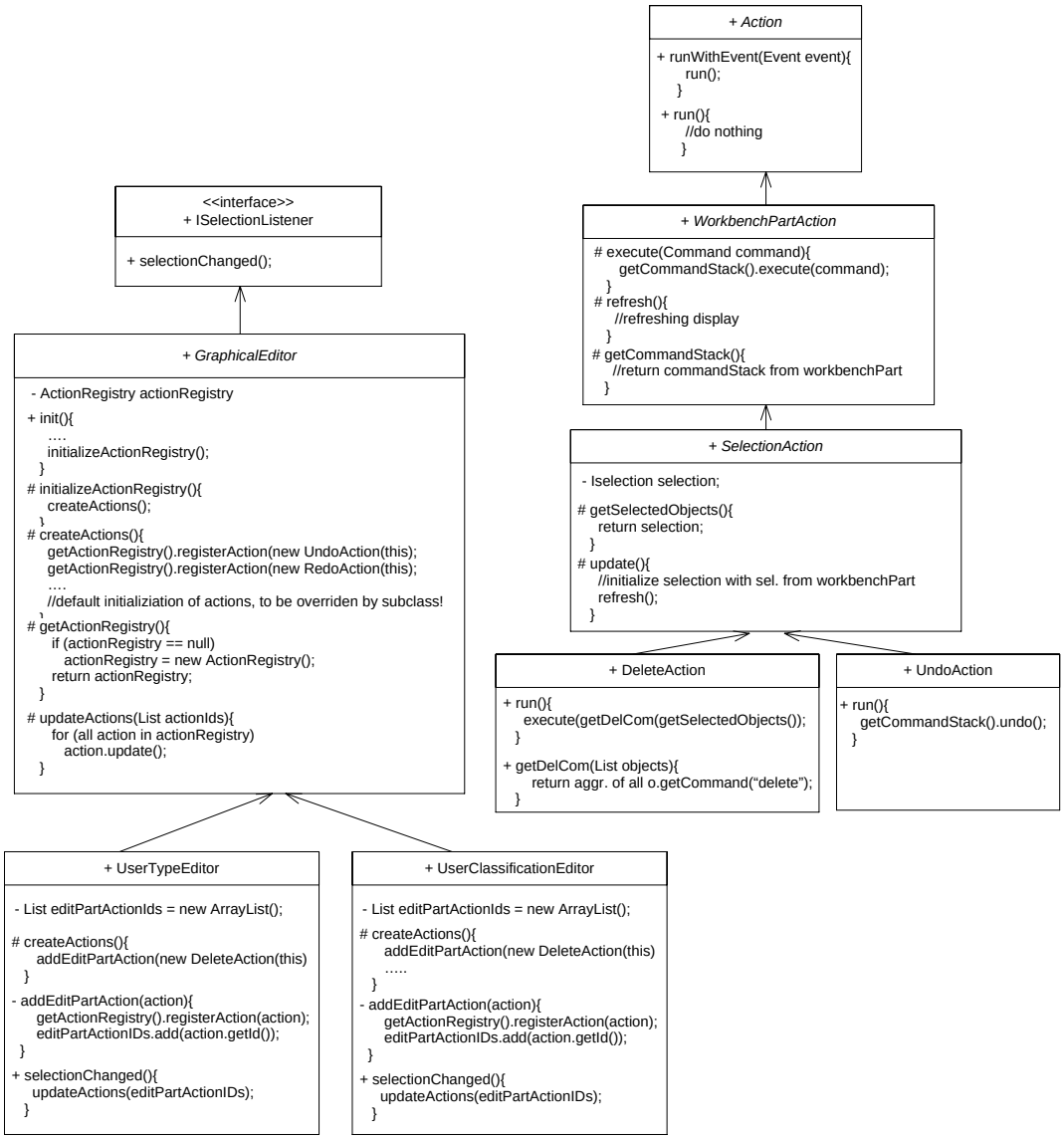

Figure 4.10: Binding a User's Interaction to a specific Command

4.5.3 Setting a Grid Layout

The `SchemaXYLayoutPolicy` extends the class `ConstrainedLayoutEditPolicy`, and `ConstrainedLayoutEditPolicy` in turn subclasses `LayoutEditPolicy`. The `LayoutEditPolicy` provides the abstract method `createChildEditPolicy`, `ConstrainedLayoutPolicy` implements this method and therein returns an instance of the GEF class `ResizableEditPolicy`. We override this method within the class `SchemaXYLayoutPolicy` and return `MoveGridEditPolicy`, a sub class of `ResizableEditPolicy`. `MoveGridEditPolicy` provides the mechanisms for arranging the schema diagram in steps of a predefined number of pixels. With its method `createChildEditPolicy`, GEF ensures to install `MoveGridEditPolicy` in each child edit part of the controller from where the `SchemaXYLayoutPolicy` was installed.

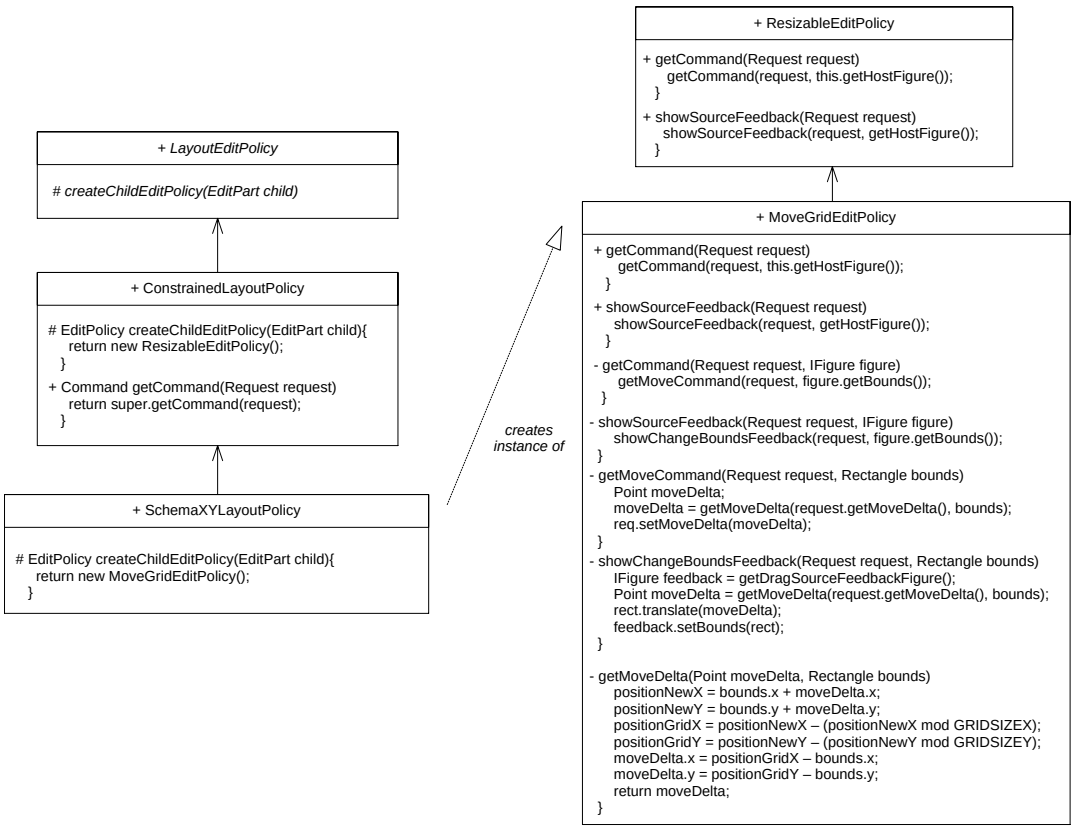

Figure 4.11: Setting a Grid Layout

Whenever the figure belonging to such a child edit part is moved, the methods `showSourceFeedback(Request request)` and `getCommand(Request request)` get called in their `MoveGridEditPolicy`. The parameter `request` has the actual type `ChangeBoundsRequest`. It particularly contains the information of how far the object is being moved both horizontally and vertically.
As it is seen from the diagram in Fig. 4.11, both methods redirect the request to methods with the same name, but additionally with a reference to the figure. We describe the two methods

in the following.

`getCommand` On the one hand, `getCommand` calls `getMoveCommand`. The task of this method is to give information about the translation coordinates to the request. This is to ensure that the figure is actually translated to the new coordinates.

`showSourceFeedback` On the other hand, we want the system to indicate an object's new position *before* we definitely decide to put it there. Therefore, `showSourceFeedback` calls the method `showChangeBoundsFeedback`. Its role is to give information on the translation coordinates to the `IFigure` feedback.
The calculation of this translation information is done with the following formulas. Moreover, it is also shown in the `getMoveDelta` method from Fig. 4.11.

$$x_{new} = x_{old} - (x_{old} \,\%\, x_{deltagrid})$$
$$y_{new} = y_{old} - (y_{old} \,\%\, y_{deltagrid})$$

4.5.4 Editing the Database

In this section we explain how we add and remove types, collections and relations from the OMS database.

Adding Types and Collections

- A new type or collection is added either by means of the OMS^{jp} Plugin or by the OMS^{jp} Schema Editor. In both cases, the wizard first of all creates the new database model objects by invoking `OMSSchema`'s methods `createObjectType(String)` or `createCollection(String, OMSType, OMSBulk)` respectively. Then, the application creates new OMS^{jp} Plugin wrapper instances of type `DBObjectType` or `DBCollection`. They obtain a reference to their original OMS object. Finally, the OMS^{jp} Plugin notification mechanism informs its listeners. Since `Schema-NotificationListener` is one of them (see Sect. 4.4.2), the GEF notification mechanism ensures that the `SchemaEditPart`, in order to update the diagram's figures, refreshes all its children controllers.

Removing Types and Collections

- Removing types and collections is a more sophisticated task than adding them. There are not only the types and collections to delete, but additionally all associated relationships, relationships partly referenced by other objects. Upon deleting collections, there are even more objects to consider. Associations and constraints have to be deleted as well.

  The first thing we do is deleting the object from the OMS^{jp} Plugin model by invoking `DBElementFolder`'s method `deleteChild(DBElement)`. Upon that event, the OMS^{jp} Plugin notifies its listeners. Finally, `SchemaNotificationListener` is the class that takes care to not only remove the *objects* from its model, but also all the associated relationships, associations and constraints. After all, `SchemaEditPart` refreshes the diagram's figures.

Adding Sub and Super Relations

- Type Editor: Adding a relation between two types is an easy task. In the database, the new relation is established by invoking the `OMSObjectType`'s method `setSuperType(OMSObjectType)`. Upon notification, the `SchemaNotificationListener` ensures to create an instance of `Relationship`. This is the special case where the OMS^{jp} Schema Editor needs to maintain a model instance that does not exist within the OMS^{jp} Plugin. In its constructor, the class `Relationship` takes care to add itself to the list of its objects' source or target relations respectively.

- Classification Editor: The procedure for the Classification Editor is the same, apart from invoking `OMSCollection`'s method `setSuperCollection` instead of `OMSObjectType`'s method `setSuperType`.

Removing Sub and Super Relations

- Type Editor: Again removing relationships is slightly more difficult than adding them. A relationship is deleted from the database by invoking `removeSuperType(OMSObjectType)` on the proper `OMSObjectType`. To update the OMS^{jp}Type Editor diagram, the difficulty is to get the proper relationship. For this reason the application inspects all target relations of the super object. The appropriate relation is found and can be deleted if the source object of this relation is equal to the sub type.

- Classification Editor: When trying to delete a relation in the OMS^{jp} Classification Editor, additionally the schema has to be searched for all the constraints associated with it. The problem here is that we cannot be sure a constraint is in either a relation's source relationship or the target relationship list. For a detailed description of this so-called 'pseudo-multi-anchor' problem, the reader is referred to Sect. 4.6.2. As GEF does not allow the construction of multiple anchors, a constraint comprising three or more relations still only has one source and one target anchor. The fact that such a constraint relation does have *all* the references to the relations being part of it does not help. There is no way for a relation to find its constraint in the source or target relation list if it is not currently set as the constraint's source or target object. Therefore the only way is to ask the `SchemaEditPart` for all the constraints, and to remove all constraints that keep a reference to the relation that is to be deleted.

  Upon clicking with the right mouse button on a relation, the relation gets marked. At first sight, it may seem to be a more elegant solution not only to mark the relation, but at the same time all the constraints that are being removed simultaneously. The problem though is that the context menu could further be extended, but marking works independently of the choice in the context menu. Bold printing the constraints will probably not be senseful when intending to execute other operations than a delete.

4.5.5 Printing

We have added a printing mechanism to our OMS^{jp} Schema Editor. The task can be divided into the two parts registering the action and setting the contributor class. We describe them in the following.

Registering the Print Action Upon instantiating the `UserClassificationEditor` or the `UserTypeEditor`, GEF calls the method `initializeActionRegistry` from their super class `GraphicalEditor`. This method in turn calls, among others, the protected method `createActions`. This method has a default implementation in the `GraphicalEditor`, but as we wanted to create our own actions, we have overriden this method within both the classification and type editors. Therein, we create a new instance of `PrintAction` and add it to the `ActionRegistry`. The diagram in Fig. 4.12 illustrates this procedure.

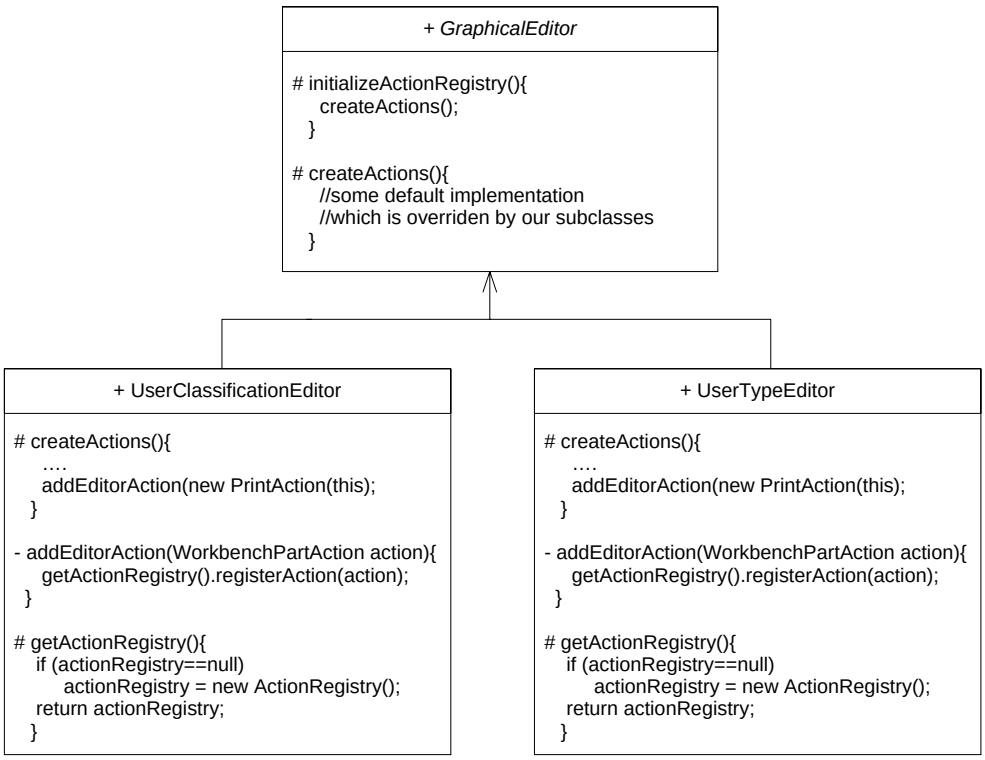

Figure 4.12: Registering the Print Action

Setting the Contributor Class As mentioned before, registering the action is not all that has to be done to make the printing mechanism work. In the file `plugin.xml`, the `SchemaActionBarContributor` is added to the editor as its contributing class. Its super class, the abstract class `ActionBarContributor`, provides the method `setActiveEditor`. By means of this method, GEF informs the `SchemaActionBarContributor` of the currently activated editor. In this way our application knows when to enable the printing icon in the tool bar and in particular which editor content to send to the printing device. Fig. 4.13 illustrates how to set the contributor class.

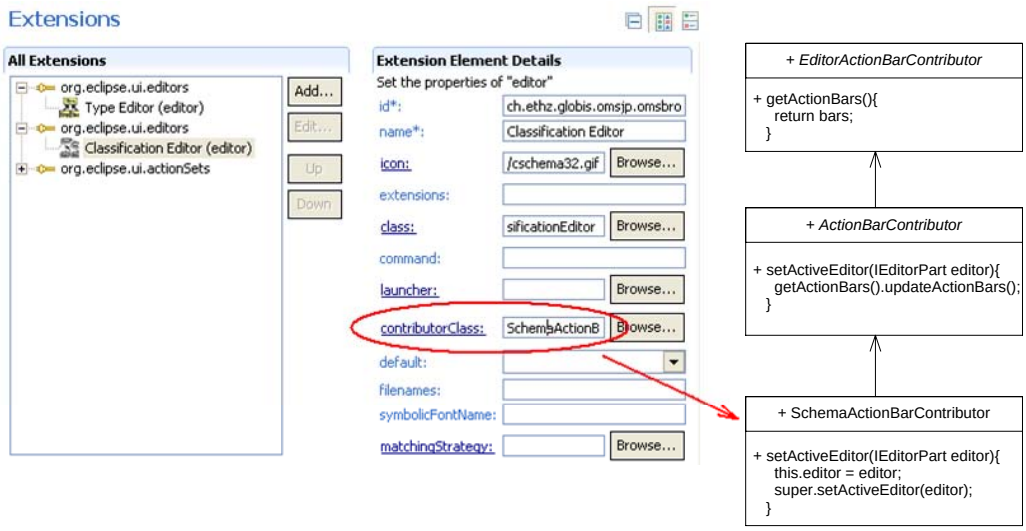

Figure 4.13: Setting the Contributor Class

4.5.6 Saving

The saving functionality for the database layout is provided by the class `GraphicalEditorWithSavingMechanism`. In the following, we describe how to activate the saving icon and the saving procedure itself one after the other.

Activating the Saving Icon The first thing to think about is *when* to activate the saving icon. The most reasonable approach is to enable the button whenever there has been a user interaction on the schema diagram.
There is a sub class of `Command` associated with each user interaction on the schema. To enable the saving icon, such a command needs to call the method `firePropertyChange` on the abstract class `WorkbenchPart`. `WorkbenchPart` is the class that has the control over the button, and a parameter 'dirty' tells the method `firePropertyChange` that the editor's contents have been changed.
The procedure how to fire a property-change event from a command in the `WorkbenchPart`is described in the following. Upon calling the `execute` method within a command (in fact in any extension of a command as the `Command` class itself is abstract), GEF goes through all the command stack listeners registered and calls the method `commandStackChanged(event)` on them. `GraphicalEditorWithSavingMechanism` has added itself to the group of listeners beforehand. It has created an instance of `CommandStack` and added itself as a listener in its `init` method. It is able to do this since its super class `GraphicalEditor` implements the interface `CommandStackListener`. After all, `GraphicalEditorWithSavingMechanism` is also a sub class of `WorkbenchPart`, and it simply needs to call `super.firePropertyChange(dirty)` within its `commandStackChanged` method. In doing so, it informs the `WorkbenchPart` of enabling the saving icon in the tool bar. The class diagram in Fig.

4.14 illustrates this mechanism.

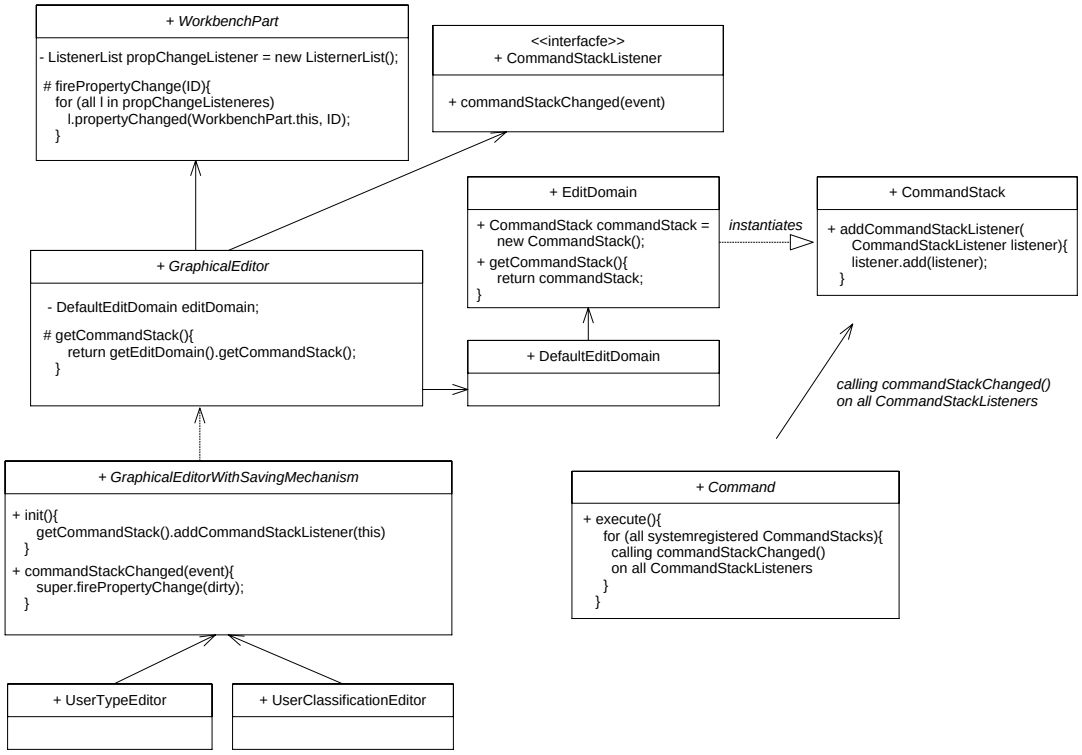

Figure 4.14: Enabling the Saving Icon

Saving Whenever a user clicks on the saving icon, the abstract method `doSave` gets called on the class `EditorPart`. `EditorPart` is not depicted in the diagram of Fig. 4.15, but it implements the `ISavablePart` interface and acts as a super class of `GraphicalEditorWithSavingMechanism`. Hence the concrete implementation of the method is offered in our class `GraphicalEditorWithSavingMechanism`. As we have seen earlier, loading the layout expects one single file in the base directory of the database. The file is supposed to consist of the database name followed by the appendix `.xml`. In order to handle all information within just one file, we have to ensure that information is not lost or overriden during the saving procedure. Therefore, the doSave method performs the following steps. First of all it creates a new instance of `XMLMemento`. Then it transfers the layout data from all other editors to the new `XMLMemento` by means of the method `restoreOldEntries`. Subsequently, by calling the method `addNewLocations`, the entries of the actual editor are joined. Finally, the `XMLMemento` is stored to the file by calling `memento.save(FileWriter)`. The procedure is depicted in Fig. 4.15.

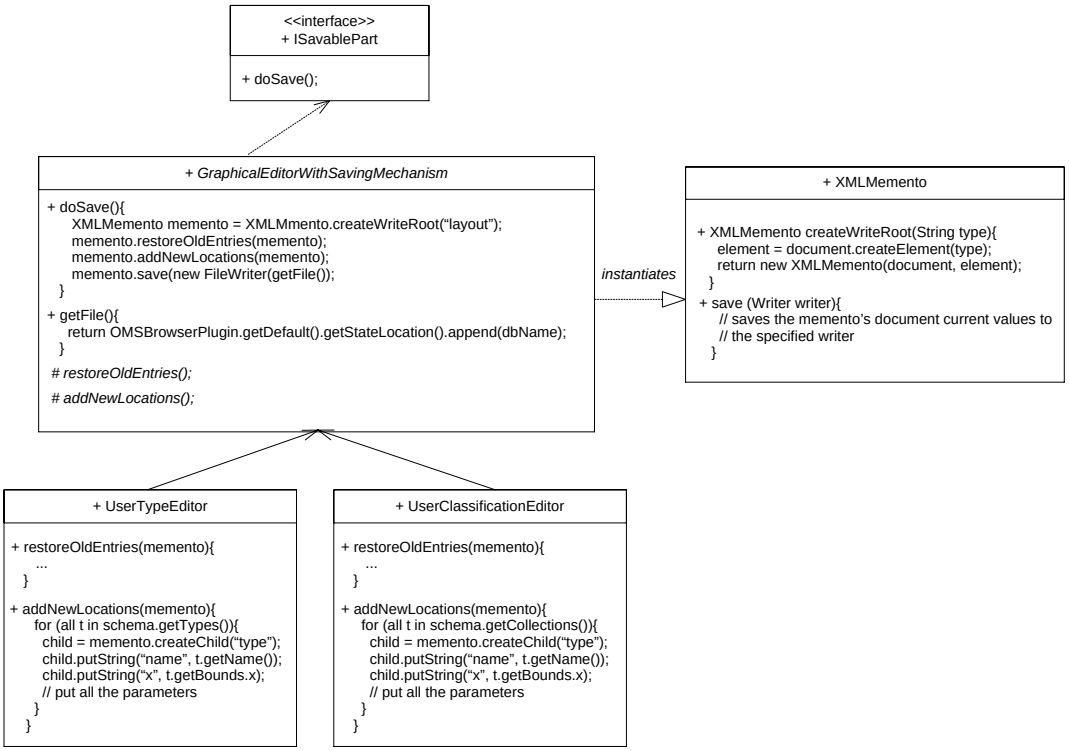

Figure 4.15: Saving Procedure

4.6 Constraints

Building constraints is a more sophisticated task than creating relationships. Constraints are not relations among normal GEF objects, but connections between relations. For this reason, the `RelationshipPart` itself has to `NodeEditPart` and returns `RelationAnchors` from its `getSourceConnectionAnchor` and `getTargetConnetionAnchor` methods.

4.6.1 Anchors

As we have seen in Fig. 4.5, `RelationAnchor` is a sub class of `AbstractConnectionAnchor`, and `AbstractConnectionAnchor` in turn implements the interface `ConnectionAnchor`. The `ConnectionAnchor` does only provide one method `getReferencePoint`. However, one and the same relation must provide anchors for both kinds of constraint arcs. That is why we introduced the methods `getReferencePointUpperCircle` and `getReferencePointLowerCircle`. Therein, the `RelationAnchor` class is able to calculate the anchor's position. It has access to the underlying polyline through its method `getOwner`, provided by `AbstractConnectionAnchor`.

4.6.2 Reasoning about a Multiple Anchor Connection

Another problem that arises upon visualising constraints is the fact that a constraint arc does not only connect *one* source and target. A constraint can be built of arbitrarily many collections. Therefore a constraint arc must be capable of having multiple anchor objects. The problem with GEF is that it only supports connections with one source and one target object. So we had to find a way to either extend GEF's connection mechanism or to accept this limitation. The first approach would result in the construction of an own `AbstractConnectionEditPart`. Not enough, even an new direct super class `AbstractGraphicalEditPart` would have to be built since `AbstractGraphicalEditPart` is the class that is responsible for setting the source and target controllers.
Moreover, a circle is completely defined by three points. For the construction of the arc, it is not clear which points to consider when we have more than three of them. For all these reasons we have decided to accept the restrictions given by GEF.
There were two possibilities now:

- Building a set of arcs out of the given anchors, each one connecting two points
- Getting always the two outermost reference points for constructing the arc

We decided for the latter. We get the centre of the arc from connecting the two reference points and taking the midpoint on this connection. By constructing a line that crosses this connection vertically in its midpoint, we are finally able to get the centre of the arc. It lies on this line at a specific distance to the crossing point. The current release of the OMS^{jp} Schema Editor takes the *exact* midpoint of the two anchors and always draws the whole halfcircle. The visualisation of an arc constructed in this way shows good results in almost all arrangements.

4.6.3 Editing Constraints

In this section we finally discuss important details to be considered when a constraint is created or deleted.

Creating Constraints The binding from the user action to a command is similarly to the way described in Sect. 4.5.2. The main difference is that the activation of a create constraint command is not linked to one single controller, but to two or more of them. Therefore the command is not generated in each `RelationEditPart`'s installed `EditPartPolicy`, but in the `Action` class itself.

Deleting Constraints Upon creating constraints in the classifcation editor model, the constraint model obtains references to the parent as well as to all the children collections. It is important that the constraint model holds these references, otherwise the user would not have the possibility to delete the constraint from the underlying database anymore.

5 Conclusions

5.1 Results and Achievements

In this section, we present the achievements that concern the common functionality, the OMS^{jp} Type Editor and the OMS^{jp} Classification Editor consecutively. The minimum requirement our OMS^{jp} Schema Editor had to fulfil was to implement the functionality that the OMS Pro Editors already provide. The consequence of our efforts is an application that implements this minimum requirement and, as it communicates over OMS^{jp}, is ready to be used by different OMS platforms. However, we did not only adopt existing functionality, but we turned our attention to the *enhancement* of the existing OMS Pro Schema Editors. This section shows these improvements on the existing editors.

5.1.1 OMS^{jp} Type and Classification Editors

Anchors As described in Sect. 2.2.2, the OMS Pro Type Editor does only know one anchor for each object, and the OMS Pro Classification Editor knows only one anchor for each edge of an object.
Within the OMS^{jp} Schema Editors, the anchors vary continuously. Reference point for a relation or association connection line is the centre of an object. A relation actually connects the centre points of the source and target objects, but the line is only drawn up to each object's border line. This object arrangement with continuous connection anchors looks much better than the arrangement with fixed anchors.

Swapping Editors The OMS^{jp} Schema Editors are clearly arranged. When swapping from one editor to another, the old one is disabled and the new one enabled, and the old editor is still reachable from its tab in the Eclipse editor area. When an editor is already opened upon clicking on its icon or choosing its specific menu entry, it is not reopened in another window. The editor rather gets reactivated. Moreover, the Eclipse editor area remains unchanged whenever a user switches perspectives.

Relations As we have seen in Sect. 2.3.2 and 2.4.2 , the OMS Editors do not allow objects to be added that are already either direct or indirect super objects. We have eliminated this weakness and work with *direct* sub and super objects within our application.

Zooming The OMS Pro Editors also provide a zooming functionality. However, the zooming mechanism affects only the objects sizes, but not the sizes of their names. Contrary, the zooming functionality of the OMS^{jp} Editors increases and decreases *all* objects as a whole, including their names. Moreover, it provides a mechanism not only to zoom in and out, but also to strictly adapt the diagram's size to the editor area concerning its width, its height or both.

5.1.2 OMS^{jp} Object Type Editor

Editing methods By double clicking on an object type or choosing the entry 'edit methods' from its context menu, the OMS^{jp} Type Editor offers a more comfortable way for adding and removing methods.

Attributes The OMS^{jp} Type Editor also provides a much more intuitive way for adding and deleting attributes.

- Adding Attributes

  An attribute is added by clicking on a type's name section and choosing the 'New Attribute' entry from the pop-up menu. In the dialog window, the user can enter a name, a type and a bulk value. Therein, he is offered the possibility to browse for the type name. The user can add as many attributes as he wants, they are *all* displayed.

- Deleting Attributes

  Deleting attributes is more intuitive. Instead of editing an attribute's name and removing all its letters, the user can simply right click on an attribute and choose the entry 'Delete' from its pop-up menu.

Furthermore, we implemented a mechanism to rearrange attributes. The rearrangement of attributes is established by holding the left mouse button down on the attribute and dragging it to the desired location.
What is not implemented yet is a way to edit existing attributes. The simple implementation without a validity check would not have been difficult. For a clear implementation however it would have been necessary to check the type names for accuracy. As the editing procedure can be replaced easily by the deletion of an attribute and the creation of a new one, editing attributes is not an indispensable task for a type editor. We therefore decided to put the focus on more relevant issues within the scope of this diploma thesis.

5.1.3 OMS^{jp} Classification Editor

Adding Collections For adding a collection, a wizard is opened where the user can type in the collection's name and browse for its type throughout the database. Alternatively, he also can enter the name manually without browsing for the type. In this case, the application checks the type name at run time and the 'finish button' is not enabled until a valid type name is entered.

Collection Names The size of a collection's name is adapted dynamically according to its length. The user is also given the possibility to alter the text size within the preference page. However, this automatic layout is already a satisfactory solution in most cases. The automatic layout feature preserves the user from having a lot of work in individually adapting the schema diagrams to his needs.

Adding Associations The wizard that is responsible for adding associations is opened by clicking on the source and target objects. The names of these objects are filled into the text fields for the source and target collections. The advantage over the OMS Pro Classification Editor is that the user is given the opportunity to enter the name and to change the default cardinalities (0,*) to arbitrary values.

Changing an Association's Name Contrary to the OMS Pro Classification Editor, changing an association's name is done without any unwanted side effects.

Constraints Unlike the OMS Pro Classification Editor, the selection of a relation for establishing a constraint is undone by simply clicking anywhere in the editor area. Furthermore, there is no circumstantial necessity to press a 'constraint icon' beforehand. The application is clever enough to offer the appropriate menu entries at the right time.

5.2 Future Work

This section shows issues that are suitable for further extension. Implementing these extensions is a challenging task. Unfortunately, they could not have been realised because of the restriction of time for this diploma thesis.

5.2.1 Copying Areas as a Whole

The generation of a database might be facilitated by offering the possibility to not only create new objects, but to copy areas of types and in particular collections. Such a copying facility is not simple to implement since there are lots of special cases to be considered. Indeed it is obvious to create a new object for each type or collection in the selected area. However, associations and relations have to be checked first. Both their source and target objects must lie in the selected region, and if they do not, the relations and associations have to be prevented from being copied.

5.2.2 Changing Object Sizes

Within the OMS^{jp} Collection Editor it might be useful to vary the objects' size. If the collection or association names get very long, the automatic font-size adaption creates very small textual name representations. Particularly association names run the risk of getting very long. However, breaking the uniformity of object sizes is not be an elegant solution either. We therefore recommend to keep the lengths of the collection, association and type names in a normal scope. A name length of up to sixteen letters should be enough for the unique classification of an object.

5.2.3 Unary Constraints

The current version of the OMS^{jp} Classification Editor could further be improved by offering the possibility to add the unary constraints ‘equal’, ‘total’ and ‘strict’.

5.2.4 Changing an Association’s Cardinality

The cardinalities of an association cannot be edited directly like the collection names can. It is more difficult to implement such a functionality for cardinalities. Its visual representation is not a feature of an object figure, but the association’s cardinality is simply represented as a label decorating the association line.

5.2.5 Placing the Objects at a Specific Location

At first sight, it might seem to be an easy task to place new objects at exactly the position where the mouse button is pressed. However, the creation wizards rely on the OMS^{jp} Plugin. These wizards do not provide the functionality for keeping information on an object’s location. The programmer of such a function should consider how to integrate this information without harming our separation property.

5.2.6 Saving and Loading Layout Preferences

We suggest a saving mechanism for the layout preferences to be a useful feature. The OMS Pro Schema Editors lack this feature too. The easiest way to fulfil this task is to rely on the existing saving procedure that uses the file `eclipseLayout.xml`, and integrating the mechanism therein. Another useful feature on the preference page would be a way for individually adapting the grid size and a toggle button for enabling and disabling the visualisation of the grid on the screen. Moreover, the determination of the variables `a` and `b` for the constraint anchors as described in Sect. 4.3.3 would be a nice feature for a preference page as well.

6 User Manual

The OMS^{jp} Schema Editor relies on functionality of the OMS^{jp} Plugin. We therefore first thought about integrating this user manual into the existing one. However, as we have built the Schema Editor as a separate unit, we finally decided also to keep the manuals separated.

6.1 Common Functionality

This section explains functionalities that are provided by both the OMS^{jp} Classification and the OMS^{jp} Type Editors.

6.1.1 Starting the Editors

The user type editor and the user classification editor are opened by clicking on their specific icon in the Eclipse tool bar. If one is not sure about which icon belongs to which editor, dragging the mouse arrow over the icon opens up a tool-tip.
In turn, the system type and the system classification editors are opened by choosing the corresponding entries in the pop-up menu from the Eclipse menu bar 'Schema Editor'

6.1.2 Printing

The Eclipse tool bar also offers an icon for opening the printing dialog. Within this dialog, the user can choose the appropriate printer driver, the range of pages to print and the number of copies. Additionally there is the possibility to set customised properties, depending on the type of driver, such as the paper size, duplex printing and its orientation, and last but not least the number of pages per sheet.

6.1.3 Saving

The arrangement of the currently active editor can be saved by clicking on the corresponding icon, representing a disk, within the Eclipse tool bar. This icon is only enabled if there has

been a change in the layout.
Note that if there is no `layout.xml` file in the root directory of the database, the system will simply position all objects at the top left corner of the editor area.

6.1.4 Selecting Areas

By means of the 'Marquee' tool from the palette, the user is able to select an area on the screen. It is then possible to perform various operations on the selected objects:

- By pressing the left mouse button and dragging the mouse arrow the user can move the objects altogether to a different location.
- By clicking the right button on the other hand, there is the possibitity to delete all the objects contained within this area.

6.2 Type Editor

In this section we describe how objects in the OMS^{jp} Type Editor are created, edited and removed.

6.2.1 Creating

New Type A new type is simply created by choosing 'Type' from the palette on the right hand side. Subsequently left clicking on any position within the editor area opens up a wizard for creating a new object type. Note that the OMS^{jp} Schema Editor does not provide a mechanism for creating instances of base types or structured types. The editors rely on the 'New OMS Object' wizard from the database explorer of the OMS^{jp} Plugin, and this wizard only supports the creation of user types. Within the wizard, the user enters the name for the new object type. Upon pushing the button 'Finish', the application creates a new type with the given name. The object editor window is automatically opened and the OMS^{jp} Schema Editor view is refreshed.
The alternative procedure for creating a new type is to right click anywhere within the database explorer and to choose 'New' → 'OMS Object Type'. The OMS^{jp} Plugin manual describes the proceeding in more detail in its section 'Browsing and Updating the Database' → 'Creating Objects'.

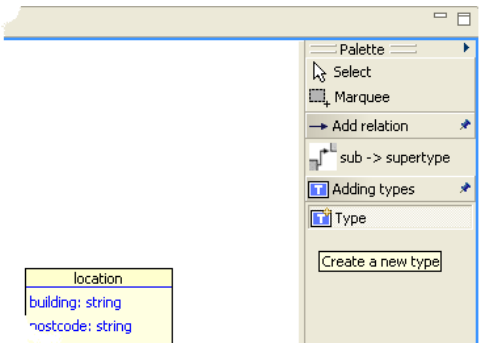

Figure 6.1: Creating a new Type

New Attribute Clicking on a specific type with the right mouse button opens a pop-up menu from where the entry 'Add Attribute' can be chosen. The user must take care not to click anywhere on the type object, but on its name area. Otherwise the pop-up menu will not contain the entries for adding a new attribute, but rather the entries for editing a particular attribute.
Selecting the 'Add Attribute' entry opens the 'Add Attribute Dialog' window from where the user can choose the attribute's name, type and bulk. Finally, the attribute is added to the type's attribute list as its last member.

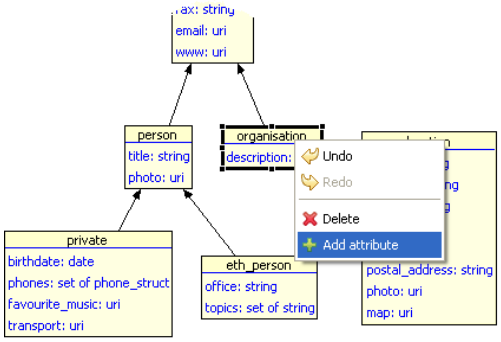

Figure 6.2: Adding a new Attribute

New Relation In order to obtain a relation between two types, the user has to open the palette and to click on the 'sub → supertype' button. The mouse arrow gets decorated by a plug indicating that the system is now ready to establish a connection between two object types. An additional symbol shows up whenever the system recognises the underlying object to be unsuitable for the creation of a connection.

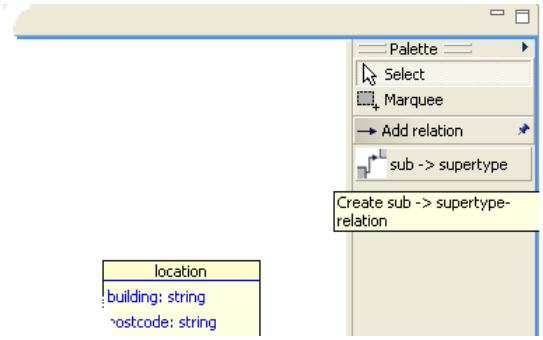

Figure 6.3: Creating a new Relation

The restrictions on building the connections are the following:

- A sub and super type must not be the same object.
- Direct super types cannot be added twice.
- The type graph is a directed acyclic graph. The user is prevented from creating cycles.

6.2.2 Deleting

Type Right clicking on a specific object type and selecting 'Delete' from its context menu removes a type from the database. Moreover, it is removed everywhere in the OMS^{jp} Plugin perspective as well as from the OMS^{jp} Schema Editor view. If the type is active as an object editor window within the OMS^{jp} Plugin environment, this window gets closed.
Note that deleting a type includes the removal of all sub and super type relationships this object type is part of.

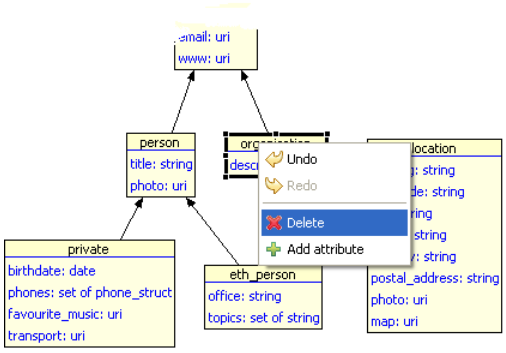

Figure 6.4: Deleting the Object Type 'organisation'

Relation A user can delete a sub or super type relation just by right clicking on the relation and choosing 'Delete'. There are no further constraints to be considered.
Deleting a relation is much more comfortable within the OMS^{jp} Schema Editor environment and is considered to be one of its strengths over the OMS^{jp} Plugin. The user now gets a visual presentation of what is happening in the underlying database.

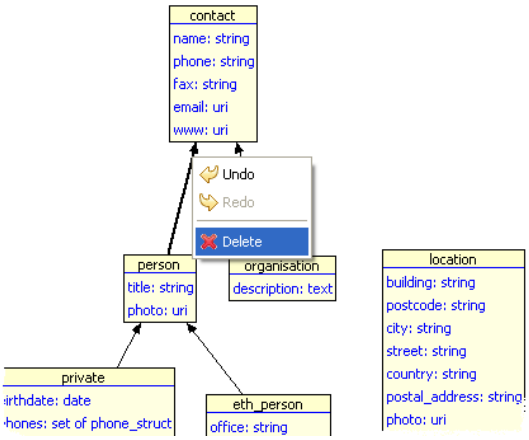

Figure 6.5: Deleting the Relation 'person' → 'contact'

Apart from this, deleting a relation is the equivalent to right clicking on a type in an object type's sub or super types section and selecting 'Remove'.

Attribute An attribute is removed very intuitively. It is deleted by right clicking on the attribute and choosing the 'Delete' entry from the context menu that appears. The result is the same as right clicking on an attribute value in the OMS^{jp} Plugin Object Editor and selecting the entry 'Remove' from its pop-up menu.

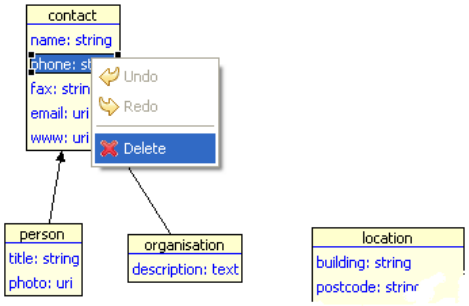

Figure 6.6: Removing the Attribute 'phone' from the Object Type 'contact'

6.2.3 Editing

Changing a Type's Name The user can alter the name of a type directly within the editor. In order to do so, he must left click on the name. Thereafter the name field gets an active bounding and the user can enter a new name. The interaction is accomplished by either pressing the return button or just by clicking anywhere in the editor area.
Changing a type's name is new feature. A user is unable to change a type's name within the OMS^{jp} Plugin.

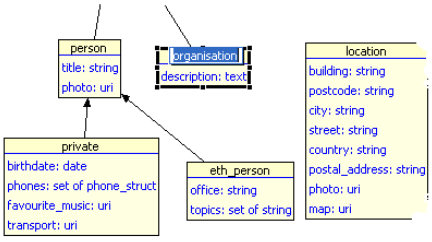

Figure 6.7: Changing the Name of the Object Type 'organisation'

Changing the Attributes' Order The attributes of a type can be arranged in an arbitrary way. A user just needs to left click on the attribute he wants to move and to drag it to the desired location within the same type. Dragging an attribute to other types is not supported in the current release.

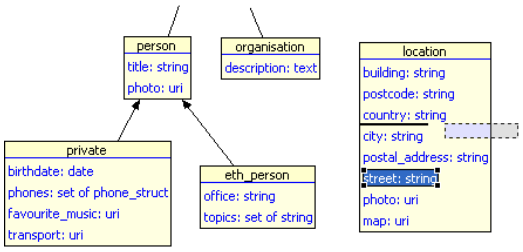

Figure 6.8: Changing the Order of Attributes

6.3 Classification Editor

Like the Type Editor section, this section also gives a description of how objects are created, edited and removed, but this time within the OMS^{jp} Classification Editor.

6.3.1 Creating

Collection A collection is created by means of the entry 'New Collection' from the palette. Subsequently clicking on an arbitrary position within the editor area opens up the collection wizard.
Within the wizard, the collection's name can be entered. Clicking on the 'browse' button opens a new window from where the according object type or base type can be chosen.

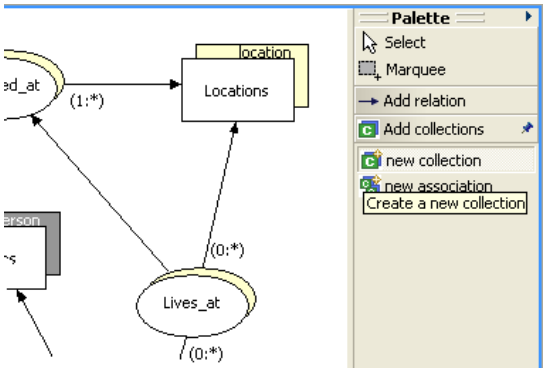

Figure 6.9: Creating Collections

New Relation The procedure is the same as within the Type Editor. The user has to open the palette and to click on the 'Sub → Supercoll' button. The mouse arrow gets decorated by a plug indicating that the system is now ready to establish a connection between objects. An additional 'not allowed'-symbol appears if the underlying object is not suitable for creating the connection.
The kind of relation is restricted by the following three items:

- The source and target collection must not be the same object.
- If a collection is already a direct super collection, it cannot be added once again.

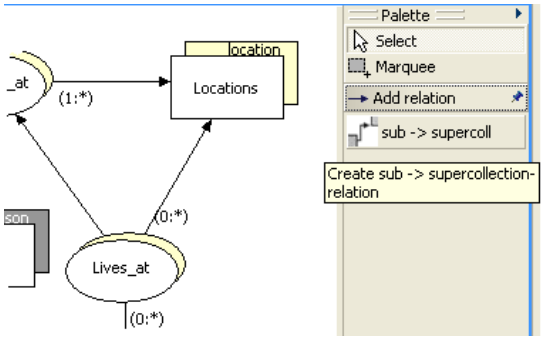

Figure 6.10: Creating new Relations

- A classification graph is a directed acyclic graph. The user is prevented from creating cycles.

Constraint Creating constraints is an event for which we have to select at least two elements in the editor area. These elements must be relations. Multiple elements are chosen by holding the 'Ctrl' button down while selecting the objects with the left mouse button.
Four different kinds of constraint are provided in the current release of the OMS^{jp} interface:

- Cover Constraint
- Disjoint Constraint
- Partition Constraint
- Intersection Constraint

There are restrictions on the object selection for all these constraints. The cover, disjoint and partition constraints can be established among arbitrarily many relations, but they must all be connecting to the same super collection. For creating an intersection constraint on the other hand the relations all must connect to the same sub collection.
Note that the three unary constraints 'equal', 'total' and 'strict' are not yet supported by the OMS^{jp} Schema Editor.

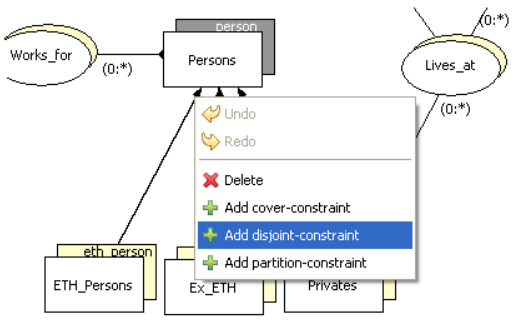

Figure 6.11: Creating a Disjoint Constraint

6.3.2 Deleting

Collection To remove a collection, simply right click on the collection and choose 'Delete' from the pop-up menu. This removes the collection from the OMS database and everywhere in the OMS^{jp} Plugin perspective. Finally, it is also removed from the OMS^{jp} Schema Editor view. If the collection is opened in a collection editor window within the OMS^{jp} Plugin environment, this window gets closed.
Note that deleting a collection includes also the removal of all sub and super collection relationships that involve this collection.

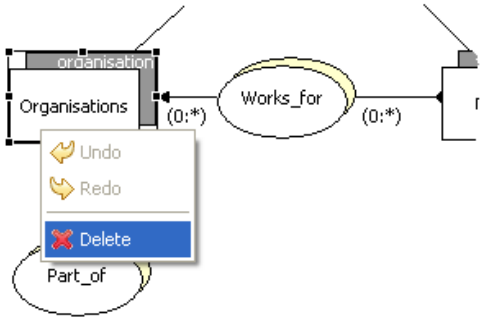

Figure 6.12: Removing the Collection 'Organisations'

Association The procedure for deleting an association is the same as the procedure for deleting a unary collection. Right clicking on the association and selecting 'Delete' in the context menu removes the association from the OMS database. Again the collection is deleted everywhere in the OMS^{jp} Plugin perspective as well as from the Schema Editor view.

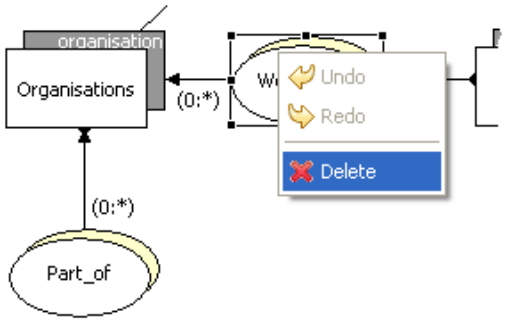

Figure 6.13: Deleting the Association 'Works_for'

Relation A sub or super collection relation is deleted by clicking on the relation and choosing 'Delete'. The procedure is depicted in Fig. 6.14.
Note that deleting a relation which is part of a constraint means that the constraints are also removed. However, these constraints are not marked when a relation is selected for deleting. The reason is that the system gets no indication whether the user really wants to *remove*

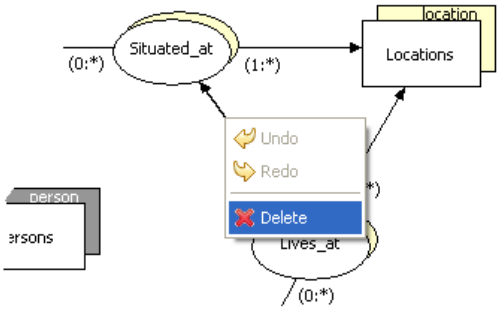

Figure 6.14: Removing the Relation 'Lives_at' → 'Situated_at'

the relation. A relation's context menu might be extended in the future to perform other operations.

Constraint The right click on a specific contsraint opens up a context menu. Selecting the entry 'Delete' removes the constraint from the database.
Constraint checking is performed at the time of a commit operation. The context menu of the OMS^{jp} Plugin database explorer offers a menu entry to commit the currently open database, and the OMS console informs the user if the commit operation has been executed successfully or not.
However, during the work with the database, the user sometimes may wish to check the constraints for consistency without actually to perform a commit. Unfortunately OMS^{jp} does not yet offer the interface to simply 'check' a database.

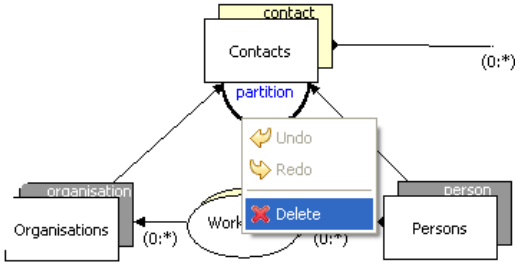

Figure 6.15: Removing a Partition Constraint

6.3.3 Editing

Changing a Collection's Name Just as type names, the user can also alter a collection's name within the editor. A single left click on the name sets an active bounding and the user is able to change the name. Again the interaction is accomplished by either pressing the return button or by just clicking anywhere in the editor area.
Changing a collection's name is also a new feature. The user is unable to change a collection's name within the OMS^{jp} Plugin.

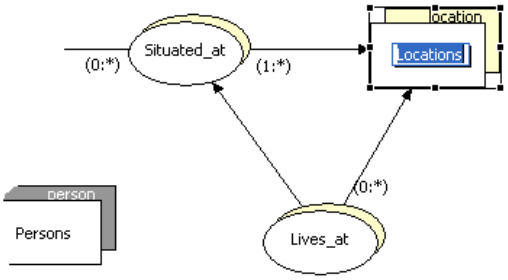

Figure 6.16: Changing the Name of a Collection

Acknowledgements

First of all I would like to thank Michael Grossniklaus for supervising my diploma thesis, for supporting me with valuable feedback and for giving me enough freedom to be inspired with new ideas.
I am also grateful to Professor Dr. Moira C. Norrie for giving me the opportunity to accomplish my diploma thesis in her group.
Last but not least I want to thank all the people who have spent time testing and evaluating my project. Their feedback gave me the chance to build an application that is reliable and, in particular, well adapted to the users' needs.

Bibliography

[1] B. Aeppli. Eclipse-Based Front-End for OMS^{jp}. February 2005.

[2] Eclipse. http://www.eclipse.org.

[3] N. Edgar, K. Haaland, J. Li, K.Peter. Eclipse User Interface Guidelines. February 2004. http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.

[4] K. Ehrig, C. Ermel, G. Taentzer. Erstellung eines grafischen editor-plug-ins mit eclipse EMF und GEF. February 2005. http://www.sigs.de/publications/os/2005/02/ehrig_ermel_OS_02_05.pdf.

[5] IBM Corp. Plugin Development Environment Guide. 2001. http://www.eclipse.org/documentation/pdf/org.eclipse.pde.doc.user.pdf.

[6] K. Griffin. Using EMF. December 2002. http://www.eclipse.org/articles/Article-Using%20EMF/using-emf.html.

[7] M. Grossniklaus. OMS^{jp}: A Uniform Java Interface to Heterogenous OMS Platforms. Technical White Paper, April 2005, Version 1.0.

[8] R. Hudson. Create an eclipse-based application using the graphical editing framework. July 2003. http://www-106.ibm.com/developerworks/opensource/library/os-gef.

[9] B. Majewski. A Shape Diagram Editor. December 2004. http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html.

[10] M.C.Norrie, A. Würgler, K. von Gunten, A. Palinginis, M. Grossniklaus. OMS Pro 2.0: Introductory Tutorial. Technical Report, March 2003.

[11] OMS - Object Model System. http://www.oms.ethz.ch.

[12] OMS^{jp} API. http://www.oms.ethz.ch/omsjp/api/omsjp.

[13] A. Würgler. OMS Development Framework: Rapid Prototyping for Object-Oriented Databases. 2000.

[14] P. Zoio. Building a Database Schema Diagram Editor with GEF. September 2004. http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html.